AFRL-RI-RS-TR-2017-176

SECURE MATHEMATICALLY-ASSURED COMPOSITION OF CONTROL MODELS

ROCKWELL COLLINS

SEPTEMBER 2017

FINAL TECHNICAL REPORT

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ AIR FORCE MATERIEL COMMAND ■ UNITED STATES AIR FORCE ■ ROME, NY 13441

NOTICE AND SIGNATURE PAGE

FOR THE CHIEF ENGINEER:

/ S /
STEVEN DRAGER
Work Unit Manager

/ S /
JOHN MATYJAS
Technical Advisor, Computing
and Communications Division
Information Directorate

REPORT DOCUMENTATION PAGE	*Form Approved* **OMB No. 0704-0188**

1. REPORT DATE *(DD-MM-YYYY)* SEPTEMBER 2017	**2. REPORT TYPE** FINAL TECHNICAL REPORT	**3. DATES COVERED** *(From - To)* AUG 2012 – APR 2017

4. TITLE AND SUBTITLE SECURE MATHEMATICALLY-ASSURED COMPOSITION OF CONTROL MODELS	**5a. CONTRACT NUMBER** FA8750-12-9-0179
	5b. GRANT NUMBER N/A
	5c. PROGRAM ELEMENT NUMBER 62303E
6. AUTHOR(S) Darren Cofer, John Backes, Andrew Gacek, Daniel DaCosta (Rockwell Collins), Michael Whalen (University of MN), Ihor Kuz, Gerwin Klein, Gernot Heiser (Data 61), Lee Pike, Adam Foltzer, Michal Podhradsky (Galois), Douglas Stuart, Jason Grahan, Brett Wilson (Boeing)	**5d. PROJECT NUMBER** HACM
	5e. TASK NUMBER RO
	5f. WORK UNIT NUMBER CK
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Rockwell Collins, 400 Collins Rd NE, Cedar Rapids, IA 52498	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Research Laboratory/RITA, 525 Brooks Road, Rome NY 13441-4505	**10. SPONSOR/MONITOR'S ACRONYM(S)** AFRL/RI
	11. SPONSOR/MONITOR'S REPORT NUMBER AFRL-RI-RS-TR-2017-176

12. DISTRIBUTION AVAILABILITY STATEMENT

Approved for Public Release; Distribution Unlimited. PA# 88ABW-2017-4397
Date Cleared: 12 SEP 17

13. SUPPLEMENTARY NOTES

14. ABSTRACT

The Secure Mathematically-Assured Composition of Control Models project (SMACCM) has developed new tools for building UAV software that is provably secure against many classes of cyber-attack. The goal of the project is to provide verifiable security; that is, system designs which provide the highest levels of confidence in their security based upon verifiable evidence. The SMACCM team has developed system architecture models, software components for mission and control functions, and operating system software, all of which are mathematically analyzed to ensure key security properties.

15. SUBJECT TERMS

Cybersecurity, software assurance, verification, formal methods

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** STEVEN DRAGER
a. REPORT U	**b. ABSTRACT** U	**c. THIS PAGE** U	NA	134	**19b. TELEPHONE NUMBER** (*Include area code*) NA

Contents

List of Figures **iv**

1 Summary **1**

2 Introduction **2**

3 Methods, Assumptions, and Procedures **4**

4 Secure Architecture **8**
- 4.1 The Assume Guarantee Reasoning Environment . . . 9
  - 4.1.1 Informal Description of Architecture and Behavior . . . 10
  - 4.1.2 ULB Modeling Assumptions . . . 11
  - 4.1.3 Scheduling Constraints . . . 13
  - 4.1.4 ULB Component Contracts . . . 15
  - 4.1.5 ULB Properties . . . 23
- 4.2 Resolute . . . 27
  - 4.2.1 The Resolute Language . . . 28
  - 4.2.2 Claims and Rules . . . 29
  - 4.2.3 Computations . . . 30

5 Secure Components **34**
- 5.1 Ivory . . . 34
- 5.2 Tower . . . 38

6 Secure Operating System **41**
- 6.1 seL4 Microkernel . . . 42
- 6.2 seL4 Verification . . . 42
  - 6.2.1 Background: Verified seL4 properties . . . 42
  - 6.2.2 ARM-hyp Verification . . . 43
  - 6.2.3 x86_64 Verification . . . 46
  - 6.2.4 WCET and Schedulability Analysis . . . 48
- 6.3 CAmkES . . . 50
- 6.4 eChronos RTOS . . . 55
- 6.5 Hardware . . . 56
- 6.6 Device Drivers . . . 56
- 6.7 File System . . . 58
- 6.8 CAN Protocol . . . 60
- 6.9 Ground Control Station Communication Protocol . . . 63
- 6.10 CAN Gateway . . . 64

7 Trusted Build **67**
- 7.1 AADL Modeling Language . . . 68
- 7.2 Code Generation . . . 69
- 7.3 Code Generator Architecture . . . 73

8 SMACCMcopter Demonstration **77**
8.1 Hardware Architecture 77
8.2 Software Architecture 78
8.3 Build Process 79
8.4 Demonstration 80
8.4.1 Successful Attack on Unsecure Vehicle 81
8.4.2 Failed Attack on Secure Vehicle 81

9 ULB Demonstration **83**
9.1 Background 83
9.1.1 Challenge Problems 83
9.1.2 Unmanned Little Bird 84
9.2 Applying HACMS Technologies to the ULB 87
9.2.1 Architecture 88
9.2.2 Components 94
9.2.3 Kernel 98
9.2.4 Build Process 99
9.3 Demonstrations 99
9.3.1 Initial Demonstration 100
9.3.2 Final Demonstration 101

10 Results and Discussion **104**
10.1 Type-Checking for Embedded Programming 104
10.2 Type-Safe System Plumbing 105
10.3 Faking a Module System 105
10.4 Control Your Compiler 106
10.5 Everything is a Library 106
10.6 Semantics 106
10.7 Integration Problems 108
10.7.1 Drivers 108
10.7.2 Legacy/Untrustworthy Code Support 109
10.7.3 Make Issues 110
10.7.4 Exceeding Static Memory when Generating System Image 110
10.7.5 C Linker and Typing 111
10.7.6 System Image Loading and Debugging 111
10.8 Modeling Concerns with AADL 112
10.8.1 IRQ/ISR Representation 112
10.8.2 Thread/Process Semantics 113
10.8.3 Events and Schedulability 113
10.9 Lessons Learned from ULB Application 114
10.9.1 Ivory 114
10.9.2 HCOL and Spiral 115
10.9.3 AADL and Trusted Build 115
10.9.4 Tool Integration 116
10.9.5 Build Times 116
10.9.6 Debugging 117

10.9.7 Development Ecosystem 117

11 Conclusion 118

12 References 119

13 List of Acronyms 125

List of Figures

1 Constraints to force variables representing fields of incoming and outgoing data the same between components 13
2 The definition of the variables for component clocks rising and falling 14
3 Constraints about when a component may begin executing . . . 14
4 Constraints about when a component may stop executing 15
5 Constraints overriding the normal semantics of the connections to the `loi` component 16
6 Definitions for the state variables of the `loi` component 17
7 Definition for the `loi_approved_for_message` component. The definition of this variable was based on the STANAG 4586 specification 18
8 The guarantees and assumptions of the `loi` component 19
9 The contract of the `authin` component 21
10 Definitions for CUCS Authorization Request and Payload Steer STANAG 4586 message types 22
11 The contract for the `input` component 23
12 Definitions for the state variables referenced by the system level guarantees 24
13 The assignment statements used to reference state variables in the top level contract 25
14 The guarantee that whenever an authorization message is received and the current LOI is 3 the vehicle accepts this message within a specified latency 26
15 The guarantee that if the vehicle input receives a navigation command then the current LOI is 4 26
16 Guarantees about the state machine in the `input` component. . . 26
17 A guarantee about mode transitions under a certain LOI 27
18 A guarantee about routes being uploaded to the aircraft 28
19 Example Resolute rules 30
20 Definition of `bound` in the Resolute standard library 31
21 A simplified picture of the software architecture for the SMACCMcopter. The red line illustrates the path that valid commands take to reach the motor controller. 32
22 Example of a successful assurance case from Resolute 32
23 Example of a failed assurance case from Resolute 32
24 Example Ivory module definition 37
25 Tower (top), Task (middle), Graphviz output (bottom) 39
26 Overview of seL4 verification. 43
27 Architecture for microkernel-based hypervisor. 44
28 Overview of dataflow in the WCET analysis. 49
29 Overview of the CAmkES build process. 51
30 CAmkES theory dependencies. 54
31 Overview of Termite synthesis process. 57

32 Code/Proof Co-Generation with Cogent. 58
33 Message passing between the different protocols/components. . . 61
34 SMACCM Build Process for eChronos 70
35 SMACCM Build Process for CAmkES/seL4 70
36 Excerpt from the SMACCMcopter Mission Software AADL model 72
37 AADL thread dispatch model 73
38 C code generated from SMACCMcopter Mission Software AADL model 74
39 Communicating AADL threads A, B and their implementation in CAmkES 75
40 Flow of interaction between A, B, and monitor M 76
41 SMACCMcopter during final flight demonstration 77
42 SMACCMcopter computing hardware: Pixhawk flight control computer (left) and TK1-SOM mission computer (right) 78
43 Software architecture for SMACCMcopter flight computer 79
44 Software architecture for SMACCMcopter mission computer . . . 79
45 Software architecture of the secure SMACCMcopter, illustrating the attack 80
46 Failed cyber-attack showing ground control station (left) and attacker laptop (right) 81
47 The Boeing Unmanned Little Bird 83
48 The Unmanned Little Bird Unmanned Aircraft System 85
49 ULB Baseline Architecture 88
50 ULB HACMS Phase 1 Architecture 89
51 ULB HACMS Phase 2 Architecture 90
52 ULB HACMS Final Architecture 91
53 Dynamic Geo-Fence Enforcement 93
54 Original C++ Code 95
55 Ivory Concrete Syntax 96
56 Generated C Code 96
57 ULB VSM Build Process 100
58 ULB Risk Reduction Flight Demonstration 24 July 2015 102
59 ULB Final Demonstration 9 February 2017 102
60 Conditional Ivory macro. 107

1 Summary

Unmanned Air Vehicles (UAVs) and other military aircraft have off-vehicle network connections for command and control, sharing sensor data, and coordination with other forces. This connectivity provides tremendous new functionality, but also exposes these military assets to the same security risks that plague laptop computers and web servers.

Current approaches to cybersecurity rely on patching systems after a vulnerability is discovered. What is needed is a clean-slate, mathematically-based approach for building secure software. DARPA initiated the High Assurance Cyber Military Systems (HACMS) program to develop the technologies needed to counter cyber-threats to network-enabled embedded systems.

The Secure Mathematically-Assured Composition of Control Models project (SMACCM) has developed new tools for building UAV software that is provably secure against many classes of cyber-attack. The goal of the project is to provide verifiable security; that is, system designs which provide the highest levels of confidence in their security based upon verifiable evidence. The SMACCM team has developed system architecture models, software components for mission and control functions, and operating system software, all of which are mathematically analyzed to ensure key security properties.

Unique aspects our approach include:

- Integration of proof and software engineering
- Formally verified operating system software
- High-assurance control components based on formal design languages and synthesis

We have prototyped these new technologies on a research quadcopter, and then transitioned them to Boeing's Unmanned Little Bird (ULB) helicopter to demonstrate their practicality and effectiveness. Our tools for secure software development and analysis and all software produced are available for transition to other vehicle programs.

2 Introduction

Embedded systems form a ubiquitous, networked, computing substrate that underlies much of society. Modern aircraft and automobiles are complex safety-critical systems in which software is an integral part of the vehicle control and functionality. Like other embedded systems, vehicles are now networked for a variety of reasons, including the ability to conveniently access diagnostic information, perform software updates, provide innovative features, lower costs, and improve ease of use.

However, researchers (and hackers) have shown that all kinds of networked embedded systems are vulnerable to remote cyber-attack. A number of hacks and hacking attempts recently reported in the press target the control systems of vehicles. For example, researchers at University of Washington and University of California San Diego have demonstrated the ability to completely control an unmodified automobile from a remote location [1]. Security researchers Charlie Miller and Chris Valasek have recently extended this work [2]. Others [3], [4] have been probing for vulnerabilities in the communication and avionics systems of commercial aircraft, though with questionable success. The consequences of a successful cyber-attack against these systems include loss of life or denial of military capabilities, and not just the compromise of classified information.

Why are embedded systems so vulnerable now? Three trends in the way vehicle control systems and embedded software of all kinds are developed have conspired to bring us to this point. First, network connectivity has been added to devices of all sorts to enable new functionality. In many cases, this addition was done *post hoc* and introduced the cyber-threats from the Internet to the embedded world – a world where in the past it was safe to assume that failures were random and independent, rather than coordinated and malicious. Second, embedded software is larger and more complex than ever. This complexity means that, all other things being equal, embedded software will contain more vulnerabilities that an adversary can exploit. Third, while in past decades most embedded software was custom-built, modern systems are developed using the same commercial and open source tools and software libraries used in traditional computing platforms. This commonality of operating systems, communication stacks, and even code generators and compilers exposes embedded software to many potential exploits.

To combat this threat, DARPA initiated the HACMS research program to create technology for the construction of cyber-physical systems that would be resilient against cyber-attacks. The program is focused on vehicle control systems (both air and ground) because of their complexity, criticality, and significance for the military and civilian worlds. High assurance is defined to mean functionally correct and satisfying necessary safety and security properties. Achieving this goal requires a fundamentally different approach from what the software community has pursued to date. Consequently, HACMS has adopted a clean-slate, formal methods-based approach to enable semi-automated code synthesis from executable, composable, formal specifications which are subject to analytic verification. The term *formal methods* refers to the analysis of software (or models of software) to prove its conformance to specifications. The use of analytic techniques for verification is, of course, standard practice in the control community, but is relatively new in the software community.

At least to some extent, we are in the current vulnerable state because we have not paid attention to cybersecurity as a design requirement for safety-critical control systems. We have allowed our software implementations to include network connectivity, grow in complexity, and use more COTS software without sufficient regard to the associated security hazards. Cybersecurity is a system property that must be designed-in from the beginning to be effective. New tools and techniques, such as those being developed in the HACMS program, can help us to build control systems that are both safe and secure.

The traditional approach to cybersecurity is reactive, responding to cyber-attacks after they occur by identifying a vulnerability and developing a software patch to eliminate that specific vulnerability. This is a cycle that repeats itself with each new vulnerability that is found. Even virus-scanning software cannot keep up with the pace of newly created malware, and in fact, often introduces new vulnerabilities that can be exploited. The situation is even worse for embedded software because it is often difficult to patch due to logical issues or certification constraints. The goal of HACMS program research is to break this cycle by preventing security vulnerabilities from being introduced during the development process.

Our project in the HACMS program, Secure Mathematically-Assured Composition of Control Models, brings together four main concepts based on formal methods. The system architecture is first modeled in a way that permits formal verification of its key security and safety properties. Software components in this architecture are then implemented using languages that guarantee important security properties. Constraints on component interaction and execution specified in the architecture model are enforced by a secure microkernel whose functional and security properties have been formally verified to the binary level. Finally, the deployed system is automatically built from the verified architecture model and component specifications.

To show that this approach is both practical and effective, we have applied it to two aircraft. We first developed the technologies on a modified commercial quadcopter that includes separate flight control and mission computers, a data bus for communication, multiple sensors, a video camera payload, and radio links for control, telemetry, and surveillance. We refer to this research vehicle as the *SMACCMcopter*. We then applied the same technologies to Boeing's Unmanned Little Bird, a full-sized optionally-crewed helicopter capable of autonomous flight. Successful flight demonstrations and security evaluations by the HACMS "Red Team" show that our approach can be used to build real systems that are resilient against most cyber-attacks.

3 Methods, Assumptions, and Procedures

The design of complex military systems such as UAVs can be greatly improved through the use of advanced modeling and analysis tools. Model-Based Development (MBD) environments such as Mathworks Simulink or Esterel Technologies SCADE have long been used in the automotive and aerospace industries. MBD supports early simulation of designs, test case generation, and automatic code generation.

Recently, formal analysis tools have been integrated in MBD environment for verification of software requirements. However, these tools still face scalability issues that limit the size of systems that can be analyzed. Furthermore, system-level descriptions of the interactions of distributed components, resource allocation decisions, and communication mechanisms are still largely captured ad hoc, rather than through the use of precise modeling tools. Application of formal analysis methods at the system level requires 1) an abstraction that defines how components will be represented in the system model, and 2) selection of an appropriate formal modeling language.

The Architecture Analysis and Design Language (AADL) has been developed to capture the important design concepts in real-time distributed embedded systems [5]. AADL is well-suited to this domain, and provides an excellent mechanism for capturing the important details of system design. The AADL language can capture both the hardware and software architecture in a hierarchical format. It provides hardware component models including processors, buses, memories, and I/O devices, and software component models including threads, processes, and subprograms. Interfaces for these components and data flows between components can also be defined. The language offers a high degree of flexibility in terms of architecture and component detail. This supports incremental development where the architecture is refined to increasing levels of detail and where components can be refined with additional details over time.

AADL has both a graphical and textual format, and tools exist for developing models in both of these formats and converting between them. The graphical format is useful for visualizing and communicating the system structure while the textual format is preferred for specification of system details and for automated analysis.

Properties are defined for AADL components to specify important configuration data such as execution period, scheduling deadlines, worst case execution time (WCET), and bus latencies. This information can be used to support computation of CPU and bus utilization and schedulability analysis. Data in AADL models provides the basis for generating configuration data for the kernel or operating system and synthesis of "glue code" that implements component interactions and access to kernel services.

In AADL, the architectural model includes component interfaces, interconnections, and execution characteristics, but not their implementation. It describes the interactions between components and their arrangement in the system, but the components themselves are "black boxes." The component implementations are described separately using model-based specification languages or traditional programming languages which are included by reference in the architecture model. This separation of implementation and architecture is an important factor in achieving scalability for the analysis tools that we have developed.

One of our core innovations is to structure verification arguments by following the AADL descriptions of the system. We do this through the use of formal *assume-guarantee* contracts that correspond to the behavioral requirements for each subsystem or component. Each component in the system model is annotated with a contract that includes the requirements and constraints that are specified and verified as part of its development process. We then reason about the system-level behavior based on the interaction of the component contracts. Contracts provide a layer of abstraction, allowing architectural-level reasoning to be done in a top-down fashion: before a component is even implemented, its contract can be used in reasoning about how the component interacts with its environment.

Next, we present an overview of the four main technologies developed under the HACMS program and how they have been integrated into a development process to produce systems that are functionally correct and free from security vulnerabilities. Each technology provides the basis for one of four key elements of *architecture-driven assurance.*

The architecture model is correct. We must first establish that the AADL architecture model is correct. The model specifies the overall organization of the system and defines the interfaces for each subsystem and component, how they interact, and what data they share. Properties to be verified can be divided into structural or behavioral properties.

Structural properties of the model can be checked statically. That is, they do not change over time. Many of the baseline AADL tools distributed with the Open Source AADL Tool Environment (OSATE) address these structural properties of the model. For example, schedulability analysis of CPUs and data busses based on allocation of computational threads and messages is performed by examining static characteristics of the model. Similarly, basic data flow analysis is accomplished by examining the connections between and within AADL components.

Behavioral properties of the system are checked using assume-guarantee contracts added to the AADL model. These contracts provide an abstraction of the implementation of subsystems and components of the model, and allow the dynamic requirements of large systems to be verified compositionally and hierarchically using our Assume Guarantee Reasoning Environment (AGREE) tool. AGREE translates the AADL structure and contracts into a collection of model checking problems that are verified or falsified with counterexamples. AGREE can also check contract consistency and realizability.

We have also developed a tool (Resolute) for constructing an *assurance case* based on the structure of the AADL architecture model. An assurance case provides the capability to address properties of the system that may be less precisely defined, or which combine verification evidence from multiples sources. Assurance cases based on the architecture model may address structural or behavioral properties, or both.

Tools developed for verifying that the architecture model is correct are described in Section 4.

The components are correct. Next, we must establish that the components have been implemented correctly. This means that they must satisfy their requirements as specified in the AADL contracts, and that they must be free from vulnerabilities that could be exploited by cyber-attackers.

Our approach is based on developing new Domain Specific Languages (DSLs) for specifying software component implementations. The new DSLs, Ivory and Tower, are embedded in the Haskell programming languages. As a result, they inherit important properties from Haskell, including *memory safety*. Once specified in the DSL, components are compiled to a restricted subset of the C programming language, so that these properties are preserved. In addition, assertions are added to the C code that check for run-time errors such as arithmetic over/underflow.

Component implementations must also be verified to satisfy their contracts with the rest of the system. This can be done by model checking, or by generating test cases from the contracts. In separate work [6], we have developed automated tools for exporting AGREE contracts from AADL into the Simulink MBD environment to support component verification by model checking. A similar approach can be used to automate verification of components implemented in other languages.

Languages and tools developed in the HACMS project for verifying the correctness of components are described in Section 5.

The system execution conforms to the architecture model. The architecture model makes both explicit and implicit statements about how the system should execute. It explicitly specifies execution times and periods for tasks, binds threads and processes to CPUs, specifies connections between components, and routes messages on communication busses. At least as important are the implicit requirements embodied in the architecture. If there is no connection defined between components, then it must be the case that no data can flow between these components. This prevents unintended access to memory across component boundaries that might be intentionally exploited by an attacker or accidentally by a faulty component.

The operating system is responsible for carrying out or enforcing all of these characteristics of the system as specified in the architecture model. Inter-estingly, the operating system is not explicitly modeled in AADL as a separate component or subsystem. Rather, it manifests itself in the specification of com-ponent execution properties, connections between components, and properties that define relationships between hardware and software in the system.

We have based much of our work in the HACMS program on the seL4 operating system kernel. Both the explicit and implicit characteristics of the system are guaranteed by the seL4 kernel. seL4 was developed with formal proofs of correctness of its functional and security properties. These proofs extend from the requirements all the way down to the seL4 binary implementation.

The seL4 operating system and its proof of correctness are described in Section 6.

The system implementation corresponds to the model. None of the previous three elements of assurance really matter unless we have confidence that the system implementation preserves the properties that have been established for the architecture and components. For this reason, it is important that our models be *generative*. This means that we can automatically generate all of the code and configuration data needed to build the system directly from the architecture and component models.

We have implemented a collection of tools and scripts called Trusted Build that allow us to build the complete binary for a system from its AADL architecture model. The AADL model specifies all the information needed to configure the seL4 operating system. It also specifies the models or source code needed to build all components in the system. Given this input, Trusted Build generates the files, configuration data, and small amounts of "glue code" needed to compile the system software. The transformations performed are simple enough that the resulting system can easily be determined to conform to the verified system models.

The Trusted Build process and tools are described in detail in Section 7.

Together, these four elements provide the basis for claiming that the system is correct and satisfies its requirements.

Another important part of this project is the demonstration of our technologies on different vehicles to show that our methods and tools are practical and effective. We have applied these technologies on a commercially available quadcopter that would be easily accessible to academic researchers. We also applied them on an actual military vehicle, Boeing's Unmanned Little Bird helicopter.

Both platforms have a similar top-level avionics architecture that is useful as a notional model for the identification of critical assets and threats. Each platform includes a real-time flight control computer for sensor processing and flight control actuation, and a mission computer that manages aircraft and mission state, radio communication with the ground control station, and hosts mission payloads such as a camera for surveillance. The flight control and mission computers communicate over an internal bus, and the mission computer communicates with the ground station over an encrypted data link.

Details about the application of HACMS technologies to these air vehicles and the flight experiments performed are provided in Sections 8 and 9.

4 Secure Architecture

The system design and its ultimate implementation in source code will be orchestrated by an AADL model and associated tools. The AADL model is also at the heart of our proof architecture: the verification of a complex system is partitioned into a series of sub-verifications integrated into the top-down decomposition of the system in AADL. By partitioning the verification effort into proofs about each subsystem within the architecture, the analysis scales to handle large system designs. The approach naturally supports an architecture-based notion of requirements refinement: the properties of components necessary to prove a system-level property in effect define the requirements for those components.

Assume-guarantee contracts provide an appropriate mechanism for capturing the information needed from other modeling domains to reason about system-level properties. In this formulation, guarantees correspond to the component requirements. These guarantees are verified separately as part of the component development process, either by formal or traditional means. Assumptions correspond to the environmental constraints that were used in verifying the component requirements. For formally verified components, they are the assertions or invariants on the component inputs that were used in the proof process.

The compositional approach we have developed is based on the use of formal assume-guarantee contracts. Each component in the system model is annotated with a contract that includes the requirements and constraints that were specified and verified as part of its development process. We then reason about the system-level behavior based on the interaction of the component contracts.

A contract specifies precisely the information that is needed to reason about the components interaction with other parts of the system. Furthermore, the contract mechanism supports a hierarchical decomposition of the verification process that follows the natural hierarchy in the system model. This is an essential aspect of our proof architecture, discussed next.

The goal of the formal analysis tools developed under the HACMS program is to give developers high confidence that the system they build accurately reflects the same system that they reason about. Our tools accomplish this by:

- Allowing users to model the system that they intend to build in a language with clear syntax and semantics (AADL)
- Analyzing this model to verify that it meets user defined specifications
- Generating the software that runs on the target platform directly from this model

This last point is important because it gives assurance that the model that was analyzed closely matches the model of the system that is built.

We have developed two different analysis tools to reason about AADL models. The first tool is the *Assume-Guarantee Reasoning Environment.* AGREE is a compositional verification tool that proves behavioral properties about AADL models using modern Satisfiability Modulo Theories (SMT)-based model checkers. The second tool is called *Resolute*. Resolute is a language and analysis tool that generates assurance cases from information embedded in the AADL models.

There is a small overlap between these tools in terms of what types of properties they can be used to prove. In general, Resolute is used to prove *structural* properties about the model. For example, one may use Resolute to express properties about what types of components are present in the model or what types of paths exist for information to traverse through the model. AGREE should be used to verify logical and behavioral properties about the system, especially those that involve state or temporal characteristics. For example, AGREE can be used to prove that a system is only ever in its initial state at power-up.

The next sections describe these two tools in greater detail, and illustrate their use in two example from the HACMS program. We first show how AGREE was used to prove that the Unmanned Little Bird obeys behavioral requirements provided by Boeing. Next we show how Resolute was used to generate an assurance case demonstrating that the SMACCMcopter only executes commands that were sent by an authorized ground station.

4.1 The Assume Guarantee Reasoning Environment

AGREE verifies properties of about a system compositionally. This approach is designed to exploit the verification effort and artifacts that are already part of typical software component verification processes. Each component in the system model is annotated with an assume-guarantee contract that includes the requirements (*guarantees*) and environmental constraints (*assumptions*) that were specified and verified as part of its development process. We then reason about the system-level behavior based on the interaction of the component contracts.

There were two objectives in using this verification approach. The first is to reuse the verification already performed on components. The second is to enable distributed, parallel development of components via *virtual integration*. In this process, we specify formal component-level requirements, demonstrate that they are sufficient to prove system guarantees, and then use these requirements as specifications for suppliers. If the suppliers' implementations meet these specifications, we have a great deal of confidence that the integrated system will work properly.

AGREE was originally developed to reason about systems that execute synchronously. These systems have straightforward translations to *Lustre*, a synchronous dataflow language interpreted by the model checkers used by AGREE. However, many systems that are modeled in AADL do not behave synchronously. Ideally one can implement a communication protocol between components, such as Physically Asynchronous Logically Synchronous (PALS) [7] that allows the abstraction of synchronous communication to be sound. However, for many systems this is not the case.

In order to model non-synchronous systems we have added additional features to the tool to allow users to place components on different clock domains. Users can then specify constraints to dictate when a component's clock may tick. Informally, the value of a component's clock affects its state in the following way:

- When a component's clock transitions from false to true (or is set to true in the initial state) the component's inputs are "latched". That is, from the component's perspective its inputs do not change until the next time its clock transitions from false to true.
- When a component's clock transitions from true to false, its state may change. The next state depends on the values of its current state, its latched input values, and the constraints given by its guarantees (provided that its assumptions have been historically satisfied).

In the remainder of this section we describe how we used AGREE to verify some important requirements for the Boeing ULB.

4.1.1 Informal Description of Architecture and Behavior

Some of the important security properties of the ULB are related to proper handling of requests from ground control stations. The ground control station for the ULB uses the Standardization Agreement (STANAG) 4586 protocol for controlling both the vehicle and its surveillance camera payload. The mission computer software includes two Vehicle Specific Modules (VSM) that handle requests from the ground. The "flight" VSM is responsible for control of the aircraft and the "camera" VSM is responsible for operation and positioning of the surveillance camera. For the properties of interest in the ULB, we will not need to model the behavior of every thread running on the mission computer. Specifically, we only modeled the behavior of the authentication components (`authin` and `authout`), the Level of Interoperability (LOI) component (`loi`), the "Input" component (`input`), and the Flight Control Computer (FCC) component (`fcc`). Informally, these five components are responsible for the following tasks:

1. `authin`: The authentication in component receives incoming STANAG 4586 messages. If the message is determined to be "valid" (it decrypts and passes authentication) then the message is forwarded to the `loi` component.
2. `authout`: The authentication out component receives STANAG 4586 messages from `loi` component and forwards them to a ground station.
3. `loi`: The LOI component receives STANAG 4586 messages from various components and forwards them to other components based on rules defined in the STANAG 4586 protocol.
4. `input`: The Input component receives STANAG 4586 messages, parses them, and then sends relevant information to the `fcc` component.

5. `fcc`: The FCC component is responsible for sending information to the FCC and receiving status information from the FCC and sending it to other components.

4.1.2 ULB Modeling Assumptions

The ULB's mission computer software was modeled in AADL and the Trusted Build tool was used to generate the actual binary that runs on the vehicle from this model. Because the software is generated from the model we have high assurance that any formal properties that we prove about the model should hold on the real system. We believe the following properties to be true of any software generated from an AADL model by the Trusted Build:

1. The only communication paths between threads running on the vehicle are present in the AADL model.
2. The AADL model accurately describes the scheduling context for each thread (the period of its dispatch).
3. The structure of the data described in the AADL model is the same as that in the running software.

AGREE attempts to formally reason about the semantics of AADL models. However, AADL is a very complex language, and defining a formal semantics for all of its constructs is exceedingly difficult. AGREE tries to strike a balance by using the scaffolding provided by AADL to constrain the communication paths between components. AGREE requires the user to specify more complex notions about a model that are hard to generally infer from AADL models. For example, AGREE assumes by default that connections between components indicate equality between the variables on the source and destination of the connection. However, AGREE does not automatically generate constraints about how threads in a system are scheduled. It is up to the user to explicitly list these constraints in the form of assertions in a component's implementation.

AGREE is also limited by the specification language it generates (Lustre) and the model checkers that it uses to prove properties. In order to tractably model the behavior of the ULB software we made several strong assumptions about software generated from an AADL model. These assumptions are spelled out implicitly by the assertions that we introduced to model the execution of the software's threads. We explicitly list the assumptions below:

1. Threads do not produce new outputs until they have completed executing.
2. Threads do not preempt each other.
3. Serialization and deserialization of messages between components is implemented correctly.
4. Data sent between components is not queued. If a new message is received it overwrites the previous message.

The first assumption is bad in general. The Trusted Build translates thread event dataports into notifications that may cause another thread to dispatch before the sending thread completes execution. Furthermore, threads that contain data ports may be preempted and have their outputs read by other threads before they complete execution.

However we feel that assumption 1 is fine for this model because the components that we are modeling only produce a single output event per dispatch. We do not need to consider executions where a component's outputs are produced at different times during execution because a component will only ever produce a single output during execution.

The second assumption is actually false. Since all of the components that we reason about in this model run at the same priority, the OS will switch between executing each currently scheduled thread based on a specified time slice. However, we believe that this assumption is sound with respect to the properties that we attempt to prove for this model. Specifically, the properties that we prove bound the amount of time it takes for events to propagate through the system. Because the execution times of each component that we modeled is orders of magnitude smaller than its period, we do not believe that our model is over constrained enough to eliminate concrete real counterexamples to these properties. We argue that if we increase the worst case execution time of each component in our model to be the sum of the worst case execution time of all threads and we are still able to prove the properties of interest, then the properties should hold for the actual software.

The third assumption allows us to more easily reason about the types of data that are transmitted between components for this model. Because the ULB mission software flattens STANAG 4586 messages into arrays before transmitting them between components, it is difficult for AGREE to reason about the structure of the data. Instead we place assertions in the main component implementation to constrain the source and destination values of these messages to be the same. For example, the `authin` component has a state variable to represent the message id of an outgoing STANAG 4586 message. We assert that the value of this variable is the same as a variable representing the message id of an incoming STANAG 4586 message in the `loi` component. This allows us to write expressions about the content of a STANAG 4586 message without fully describing the algorithm for serializing and deserializing the message in AGREE. This assumption about equality of these interface variables between components is made explicit in the AGREE annex of the process implementation. These assertions are shown in Figure 1.

The final assumption is not true for the real system, but there is no good way to model queuing behavior in AGREE. We feel that the results that the tool produces still gives us some assurance of correctness even if this assumption is not true.

In the remainder of this section we discuss 1) the constraints that are used to model the scheduling behavior of each component, 2) the contract of each component, and 3) the properties that we have formalized and checked.

```
--constrain the interfaces between components
assert loi.auth_in = authin.auth_in;
assert loi.mid = authin.stanag_mid;
assert input.stanag_mid = loi.mid;
assert loi.to_auth_stanag_mid = authout.stanag_mid;
assert input.sender_mid = loi.from_sender_mid;
```

Figure 1: Constraints to force variables representing fields of incoming and outgoing data the same between components

4.1.3 Scheduling Constraints

When the Trusted Build tool generates a binary from an AADL model it uses certain annotations in the form of AADL properties to determine how to schedule the threads present in the model. However, AGREE does not use these annotations to automatically generate constraints for the clocks associated with each thread. Instead a user must manually assert constraints about how the system executes. There are two reasons why AGREE does not automatically generate these constraints:

- The exact semantics of the system may be difficult to or impossible to model accurately in AGREE. This is primarily true for components that have more than one dispatch. AGREE implicitly assumes that each AADL component has a single thread of execution. However, the Trusted Build generates a thread for each input event port (and input event data port) when a dispatch is specified for that port.
- A user may wish to express a set of constraints that is more *abstract* than the true scheduling semantics. This can make it easier to prove properties that are true for both the abstraction and the true concrete executions of the model

The constraints shown in Figures 2, 3, and 4 were used to model the scheduling semantics of the components running on the mission computer. In order to simplify the specification we introduced Boolean variables with suffix `_clk_rise` and `_clk_fall` to represent when the clock of a component has a rising or falling edge. The definition of these variables is shown in Figure 2.

The node `rise` defined in the `AgreeTypes` package evaluates to true if and only if its input was false on the previous step and true on the current step. Similarly the node `fall` defined in the `AgreeTypes` package evaluates to true if and only if its input was true on the previous step and false on the current step. On the initial state `rise` evaluates to true if its input is true and `fall` evaluates to true if its input is false.

Figure 3 shows the constraints that we used to dictate when a component may begin executing (when its clock rises). Threads modeled in AADL may dispatch under two conditions: 1) the thread receives an event on an input event

```
eq authin_clk_rise : bool = AgreeTypes.rise(authin._CLK);
eq authout_clk_rise : bool = AgreeTypes.rise(authout._CLK);
eq loi_clk_rise : bool = AgreeTypes.rise(loi._CLK);
eq input_clk_rise : bool = AgreeTypes.rise(input._CLK);
eq fcc_clk_rise : bool = AgreeTypes.rise(fcc._CLK);

eq authin_clk_fall : bool = AgreeTypes.fall(authin._CLK);
eq authout_clk_fall : bool = AgreeTypes.fall(authout._CLK);
eq loi_clk_fall : bool = AgreeTypes.fall(loi._CLK);
eq input_clk_fall : bool = AgreeTypes.fall(input._CLK);
eq fcc_clk_fall : bool = AgreeTypes.fall(fcc._CLK);
```

Figure 2: The definition of the variables for component clocks rising and falling

```
--    non-periodic components
assert loi_clk_rise =
  ((event(authin.stanagout) and authin_clk_fall) or
    (event(input.sender) and input_clk_fall)
  );
assert authout_clk_rise = (event(loi.loi2auth) and loi_clk_fall);

--    periodic components
assert condition authin_clk_rise occurs each VSMPkg.commsec_bound;
assert condition input_clk_rise occurs each 100000.0;
assert condition fcc_clk_rise occurs each 100000.0;
```

Figure 3: Constraints about when a component may begin executing

(or event data) port or 2) the thread is dispatched by a periodic timer. The first assertion in Figure 3 constrains the `loi` thread to begin executing if and only if the `authin` component sends a STANAG 4586 message to the `loi` component or the `input` component sends a STANAG 4586 message to the `loi` component. The second assertion constrains the `authout` component to only begin executing when it receives a STANAG 4586 message from the `loi` component. The third assertion forces the `authin` component to run periodically with a constant time bound[1]. This bound is the assumed frequency of incoming messages. The final two assertions constrain the `input` and `fcc` components to run periodically at a rate of 100ms.

The constraints in Figure 3 assert that the components must run when certain events occur (either a periodic dispatch occurs or an event arrives on an input). To model the components' minimum and maximum execution times we make the additional assertions shown in Figure 4.

The first five assertions guarantee that a component will complete execution

[1]Initially we assumed these messages would arrive periodically. We can change this pattern to assume that messages arrive sporadically (with some bound) and still prove the properties that we are interested in.

```
assert whenever input_clk_rise occurs input_clk_fall occurs during [10.0, 50.0];
assert whenever fcc_clk_rise occurs fcc_clk_fall occurs during [10.0, 50.0];
assert whenever authin_clk_rise occurs authin_clk_fall occurs during [10.0, 50.0];
assert whenever authout_clk_rise occurs authout_clk_fall occurs during [10.0, 50.0];
assert whenever loi_clk_rise occurs loi_clk_fall occurs during [10.0, 50.0];

assert whenever authout_clk_rise occurs authout._CLK holds during [0.0, 10.0);
assert whenever loi_clk_rise occurs loi._CLK holds during [0.0, 10.0);
```

Figure 4: Constraints about when a component may stop executing

during the interval specified by its minimum and maximum execution times. However, for the `authout` and `loi` components we must also assert that their clocks remain high until at least their minimum execution time. This is because the semantics of the pattern used in this assertion does not constrain the second event to *only* occur during the specified interval. However, for the `input`, `fcc`, and `authin` components the periodic constraints listed in Figure 3 implicitly prevent the clock from rising again before its minimum execution time[2].

Because the `loi` component has two possible dispatches we need to add a couple other constraints to change the typical behavior of its input event data ports. These constraints are shown in Figure 5.

The AGREE connection statements override the normal semantics of the specified AADL connection. Normally AGREE asserts the equality of the variables on the source and destination of each connection. The connection statements in Figure 5 force the event variables associated with the `loi.auth2loi` and `loi.sender` inputs to be cleared whenever the `loi` component finishes executing. Without these connection constraints the contract of the `loi` component may behave as if it was dispatched by the wrong component.

Note that we have not included any constraints that prevent components from executing simultaneously. Clearly since the software is only running on a single processor components cannot execute in parallel. This constraint could be expressed by creating a variable that non-deterministically decides which component is running (assuming it has be scheduled to run and has not completed executing). Additional constraints would also need to be added to keep track of how much execution time elapses while a component is running. While feasible, it would be very difficult to express these constraints correctly. It would also be more difficult to reason about this more complex set of constraints. The weaker set of constraints that we have discussed so far in this section are strong enough to prove the properties that we are interested in.

4.1.4 ULB Component Contracts

Next we describe the assume-guarantee contracts for several critical components.

The LOI component. The AGREE annex in the LOI component houses the most detailed contract. The assumptions and guarantees that are present in the

[2]This is only true because the period is larger than the minimum execution time

```
--auth1: port authin.stanagout -> loi.auth2loi;
connection auth1 :
  authin.stanagout = loi.auth2loi and
  event(loi.auth2loi) =
    (if authin_clk_fall then
      event(authin.stanagout)
    else if loi_clk_fall then
      false
    else
      event(authin.stanagout) -> pre(event(loi.auth2loi)));

--vin1: port input.sender -> loi.sender;
connection vin1 :
  event(loi.sender) =
    (if input_clk_fall then
      event(input.sender)
    else if loi_clk_fall then
      false
    else
      event(input.sender) -> pre(event(loi.sender)));
```

Figure 5: Constraints overriding the normal semantics of the connections to the `loi` component

process were derived from the STANAG 4586 specification and from discussions with Boeing. The LOI component is responsible for keeping track of the current LOI and the Common Unmanned Control System (CUCS), or ground station that is in control of the vehicle. It is also responsible for forwarding STANAG 4586 messages to the correct components in the system.

Figure 6 shows definitions for the state variables used in the LOI component contract. In Figure 7 we define the variable `loi_approved_for_message` to be true if the current message type is correct with respect to the LOI that has been authorized. Whether or not a specific message ID is authorized at a particular LOI level comes directly from the STANAG 4586 specification. This variable is used to determine whether or not a message is routed to either of the VSMs or if the message is ignored. In total, the LOI component makes three guarantees, as shown in Figure 8.

1. If no message is received, or the message that is received is not an authorization request, then all of the LOI state variables remain the same. This property is likely implicit to any software implementation of the component, but we must make it explicit or else the model checker will choose non-deterministic values for these variables.

2. If a message is received and it is an authorization request, then it is handled according to the STANAG 4586 specification. Specifically this guarantee

```
--this represents the state of the variable that comes out of the
--auth_in component
agree_input auth_in : AgreeTypes::STANAG_4586_message.cucs_auth_req;
--this represents the message id from the autho component
agree_input mid : int;

agree_input from_sender_mid : int;

eq loi : int;
eq id_in_control : int;
eq none_in_control : bool;
eq control_station : int;
eq to_auth_stanag_mid : int;

--in the actual implementation this should be ascertained
--by checking to see if the CUCS is in the auth_map with
--a specified loi
eq is_auth_loi : bool;

eq loi2 : bool = loi = 2;
eq loi3 : bool = loi = 3;
eq loi45 : bool = loi = 4 or loi = 5;

eq prev_loi : int = 0 -> pre(loi);

eq initial_state : bool =
    loi = 0 and
    prev_loi = 0 and
    none_in_control = true and
    control_station = 0;
```

Figure 6: Definitions for the state variables of the `loi` component

```
eq loi_approved_for_message : bool = (
  if(mid = 1) then --note that this is an auth request, so we don't route it
    false
  else if(mid = 20 or mid = 21) then
    loi2 or loi3 or loi45
  else if(40 <= mid and mid <= 46) then
    loi45
  else if(mid = 47) then
    loi3 or loi45
  else if(mid = 100) then
    loi45
  else if(mid = 101) then
    loi2 or loi3 or loi45
  else if(102 <= mid and mid <= 108) then
    loi45
  else if(200 <= mid and mid <= 206) then
    loi3
  else if(mid = 207) then
    loi3 or loi45
  else if(300 <= mid and mid <= 306) then
    loi2 or loi3
  else if(mid = 307 or mid = 308) then
    loi3
  else if(400 <= mid and mid <= 404) then
    loi2 or loi3 or loi45
  else if(500 <= mid and mid <= 503) then
    loi2 or loi3 or loi45
  else if(mid = 600) then
    loi3 or loi45
  else if(mid = 700) then
    loi3 or loi45
  else if(800 <= mid and mid < 1000) then
    loi45
  else if(mid = 1000 or mid = 1001) then
    loi3 or loi45
  else if(mid = 1100 or mid = 1101) then
    loi3 or loi45
  else if(1200 <= mid and mid <= 1203) then
    loi2 or loi3 or loi45
  else if(1300 <= mid and mid <= 1303) then
    loi2 or loi3 or loi45
  else if(1400 <= mid and mid <= 1402) then
    loi2 or loi3 or loi45
  else if(mid = 1403) then
    loi3 or loi45
  else if(mid = 1500 or mid = 1501) then
    loi45
  else if(mid = 1600) then
    loi45
  else if(mid = 2000) then
    loi45
  else
    false);
```

Figure 7: Definition for the `loi_approved_for_message` component. The definition of this variable was based on the STANAG 4586 specification

```
initially:
  not event(loi2auth) and
  not event(loi2vehicle) and
  not event(loi2wescam) and
  loi = 0 and
  none_in_control = true and
  control_station = 0 and
  prev_loi = 0;

guarantee "No message recieved behavior":
--keep this up to date as new message types are implemented
   (not event(auth2loi) or mid != 1) => (
     initial_state ->
     loi = pre(loi) and
     id_in_control = pre(id_in_control) and
     none_in_control = pre(none_in_control) and
     control_station = pre(control_station)
   );

guarantee "CUCS Authorisation Requestion Behavior" :
  event(auth2loi) and mid = 1 => (
    --auth request is message #1
    if(auth_in.csm = 0 and id_in_control = auth_in.cucsid) then
    --relenquish control
      none_in_control = true and
      loi = 0
    else if ((auth_in.csm = 1 or auth_in.csm = 2) and auth_in.rloi > 3) then
    --request control w/ greater loi
      id_in_control = auth_in.cucsid and
      none_in_control = false and
      control_station = auth_in.cs and
      loi = auth_in.rloi
    else
      initial_state ->
      id_in_control = pre(id_in_control) and
      none_in_control = pre(none_in_control) and
      loi = pre(loi) and
      control_station = pre(control_station)
  );

guarantee "message routing" :
  if(event(auth2loi) and loi_approved_for_message) then
    if(control_station = 1 and loi = 3) then
      event(loi2wescam) and
      not event(loi2vehicle)
    else
      not event(loi2wescam) and
      event(loi2vehicle)
    else
      not event(loi2wescam) and
      not event(loi2vehicle);

guarantee "acknowledge authorised 800 series and 2000 messages" :
  (event(auth2loi) and
  loi_approved_for_message and
  ((800 <= mid and mid < 900) or mid = 2000)) =>
    (event(loi2auth) and to_auth_stanag_mid = 1400);

guarantee "forward messages from the sender input to the auth" :
  event(sender) =>
    event(loi2auth) and to_auth_stanag_mid = from_sender_mid;

guarantee "nothing is sent to auth if nothing is received":
  not (event(auth2loi) or event(sender)) => not event(loi2auth);

assume "valid auth data" :
  event(auth2loi) =>
    (0 < auth_in.rloi and auth_in.rloi <= 5 and
    0 <= auth_in.csm and auth_in.csm <= 2 and
    0 < auth_in.cucsid and auth_in.cucsid < 255 and
    --right now we model just two control stations
    0 <= auth_in.cs and auth_in.cs <= 1);
```

Figure 8: The guarantees and assumptions of the `loi` component

covers the following scenarios[3]

(a) If the CUCS who is in control is relinquishing control, then no one is overriding control, no one is in control, and the LOI is set to zero[4].

(b) If a CUCS is requesting control or attempting to override control and the previous LOI is 3 and the requested LOI is greater than 3, then the CUCS is granted control.

(c) If a CUCS is requesting control and no CUCS is currently overriding control, then the CUCS is granted control.

(d) If a CUCS is attempting to override control and no CUCS is currently overriding control, then the CUCS overrides control.

3. If the message is received and the current LOI is approved for the message type, then it is forwarded to the appropriate VSM. If the LOI is 3 and the control station is set to the camera VSM, then the message is forwarded to the camera VSM.[5]. Otherwise, the message is forwarded to the flight VSM. If no message is received or if the message is not approved at the current LOI then the message is not forwarded to either VSM.

These guarantees could possibly be broken out into smaller requirements rather than large nested "if then else" blocks. This is more of a choice of style and readability. The LOI component assumes that the data fields for authorization messages are in their correct ranges. This assumption should be satisfied by guarantees from the `authin` component.

The LOI component implementation provided by Boeing includes logic for searching through a list of pre-authorized CUCS to determine whether or not the requesting CUCS can obtain a certain LOI. We did not explicitly model this with our requirements. Although, it could be trivially added by creating a boolean variable that indicates whether or not the requesting CUCS was present in the list and has permissions for the requested LOI. One could even model the list in AGREE if desired. The best solution might be to just include this "CUCS is allowed requested LOI" variable in the guarantees and then make it an obligation on the component designer that lookup works correctly.

The Authentication In component. The contract for the `authin` component is shown in Figure 9. The component guarantees that it only produces a STANAG 4586 messages on its output if it received a commsec message on its

[3]The described scenarios for allowing control override are as described in the STANAG 4586 documentation. However, Boeing does not implement this standard as described. They allow any CUCS to override accesses as long as it meets the correct LOI requirements. We have simplified this guarantee to meet Boeing's requirements.

[4]Setting the LOI to zero is not explicit in the specification, but it needs to be set to some non-permissive value.

[5]These restrictions about the control station are based on Boeing's requirements and not part of the STANAG 4586 specification. The specification states that camera control commands require LOI of 3.

```
eq auth_in : AgreeTypes::STANAG_4586_message.cucs_auth_req;
eq stanag_mid : int;

initially:
  not event(stanagout);

guarantee "we only send a message out if we get one in" :
  event(stanagout) => event(commsecin);

guarantee "valid auth data" :
  (0 < auth_in.rloi and auth_in.rloi <= 5 and
   0 <= auth_in.csm and auth_in.csm <= 2 and
   0 < auth_in.cucsid and auth_in.cucsid < 255 and
   --right now we model just two control stations
   0 <= auth_in.cs and auth_in.cs <= 1);
```

Figure 9: The contract of the `authin` component

input. This restricts the component to only output STANAG 4586 messages if it just received a commsec message. The component also guarantees that any authorization message that it passes on to the LOI component has valid data. By valid data we mean that specific fields in an authorization message are within the ranges specified by the STANAG 4586 specification. This is needed to prove the assumption listed in the `loi` component. We use an `initially` statement to say that before the component's clock ticks it has no events being sent on its STANAG 4586 output.

We represented different STANAG 4586 message types by including multiple subcomponents within the STANAG 4586 message data implementation. This is shown in Figure 10. Implementing the message data this way is similar to how someone would implement it as a structure in the C language using a union for different structures over the message data field. In the contract for the `authin` component we use the variable `auth_in` to specifically reference this portion of the STANAG 4586 message data field.

The Input component. The `input` component is responsible for decoding STANAG 4586 messages and forwarding commands to the FCC component. It also determines which "mode" the ULB is in. The `input` component transitions to various modes based on the different STANAG 4586 messages it receives from the `loi` component and information it receives about the state of the vehicle from the FCC. The `input` component also reports some status messages back to the ground station via the `loi` component.

The contract of the `input` component is shown in Figure 11. We have introduced AGREE variables in the `input` component's contract to model the message ID of incoming STANAG 4586 messages, the vehicles mode, a status flag indicating whether or not a waypoint was sent to the `fcc` component, the

```
data STANAG_4586_message
end STANAG_4586_message;

data implementation STANAG_4586_message.i
  subcomponents
    m_id : data Base_Types::Integer;
    m_data : data STANAG_4586_message_data.i;
end STANAG_4586_message.i;

data STANAG_4586_message_data
end STANAG_4586_message_data;

data implementation STANAG_4586_message_data.i
  subcomponents
    cucs_auth_req : data STANAG_4586_message.cucs_auth_req;
    payload_steer : data STANAG_4586_message.payload_steer;
end STANAG_4586_message_data.i;

data implementation STANAG_4586_message.cucs_auth_req
  subcomponents
    tstamp : data Base_Types::Integer;
    vid : data Base_Types::Integer;
    cucsid : data Base_Types::Integer;
    vtype : data Base_Types::Integer;
    vsubtype : data Base_Types::Integer;
    dlid : data Base_Types::Integer;
    rloi : data Base_Types::Integer;
    cs : data Base_Types::Integer;
    csm : data Base_Types::Integer;
    wait : data Base_Types::Integer;
end STANAG_4586_message.cucs_auth_req;

data implementation STANAG_4586_message.payload_steer
  subcomponents
    tstamp : data Base_Types::Integer;
    vid : data Base_Types::Integer;
    cucsid : data Base_Types::Integer;
    station_num : data Base_Types::Integer;
    azimuth : data Base_Types::Float;
    elevation : data Base_Types::Float;
    hfov : data Base_Types::Float;
    vfov : data Base_Types::Float;
    hsr : data Base_Types::Float;
    vsr : data Base_Types::Float;
    lati : data Base_Types::Float;
    long : data Base_Types::Float;
    alti : data Base_Types::Integer;
end STANAG_4586_message.payload_steer;
```

Figure 10: Definitions for CUCS Authorization Request and Payload Steer STANAG 4586 message types

message id of outgoing STANAG 4586 messages, and a status flag indicating whether or not an uploaded route is accepted.

The guarantees of the component contract describe the state transitions that the component makes as well as when information is forwarded to the `fcc` and `loi` components.

```
--TODO fill in the logic of this component
agree_input stanag_mid : int;
eq mode : int;
eq waypoint_sent_to_fcc : bool;
eq sender_mid : int;
eq route_accepted : bool;

initially:
  mode = VSMPkg.NO_MODE and
  not waypoint_sent_to_fcc;

  guarantee "initially the vehicle starts in NO_MODE":
    (mode = VSMPkg.NO_MODE) -> true;

  guarantee "the vehicle never transtions back to NO_MODE":
    true -> not pre(mode = VSMPkg.NO_MODE) => not (mode = VSMPkg.NO_MODE);

  --this guarantee just abstracts the meaning of a waypoint being sent to the fcc
  guarantee "a waypoint message is only sent to the fcc if something is sent to the fcc" :
    waypoint_sent_to_fcc => event(send2fcc);

  guarantee "whenever a waypoint is sent to the fcc an acknowledgement is sent to the loi" :
    waypoint_sent_to_fcc => sender_mid = 900 and event(sender);

  guarantee "a received route is always accepted" :
    route_accepted = (event(loi2vehicle) and stanag_mid = 801);

  guarantee "if we transition to MANUAL WAYPOINT MODE it is because we saw certain message ids" :
    true -> (mode != pre(mode) =>
      (event(loi2vehicle) and stanag_mid = 42) or
       mode = VSMPkg.WAYPOINT_MODE and pre(mode) = VSMPkg.LAUNCH_MODE);
```

Figure 11: The contract for the `input` component

The FCC and Authentication Out components. The contracts for the `fcc` and `authout` components do not contain any assumptions nor any guarantees. However, we have defined several AGREE variables in the contracts to represent state variables of the components. This allows us to specify guarantees in the top level contract about when data is sent from and arrives at these components.

4.1.5 ULB Properties

Many of the properties of the system reference state variables of the system's components (such as the mode variable in the `input` component). In order to reference these variables in the system level contract we have created a number of equations variables that are asserted to be equal to component state variables in the system implementation. These variables are listed in Figure 12. The assignment statements used to define in the system implementation are shown in Figure 13.

```
-- state variables
eq current_loi : int;
eq loi_control_overriden : bool;
eq cucs_id : int;

eq stanag_message_received : bool;
eq incoming_stanag_mid : int;
eq auth_in : AgreeTypes::STANAG_4586_message.cucs_auth_req;

eq vehicle_received_stanag : bool;
eq vehicle_stanag_mid : int;

eq stanag_out_receives_ack : bool;
eq mode : int;
eq fcc_mode : int;
eq eo_ir_sensor_valid : bool;
eq waypoint_sent_to_fcc : bool;
eq stanag_out_receives_900 : bool;

eq input_accepts_route : bool;

eq is_hovering : bool;

eq speed : real;
eq altitude : real;

eq in_air : bool;
```

Figure 12: Definitions for the state variables referenced by the system level guarantees

Provable Guarantees. The first property that we prove about the system is that "whenever an authorization message is received and the current LOI is 3 the vehicle accepts the message within a specified latency." The formalization of this property is shown in Figure 14. An authorization message is STANAG 4586 message with a message ID of 1. We define the time in which an authorization message is received as the time that the `authin` finishes executing and forwards a STANAG 4586 message to the LOI component with a message ID of 1. The current LOI is the value of the current LOI in the LOI component at the time the authorization message is received. We consider an authorization message to be accepted if the LOI component changes the current LOI and the ID of the CUCS in control to be the values requested in the authorization message. In order to prove this property we set the specified latency to 200ms.

The second property that we prove is shown in Figure 15. This guarantee states that if a navigation command reaches the `input` component then the

```
assign current_loi = loi.loi;

assign auth_in = authin.auth_in;

assign cucs_id = loi.id_in_control;

assign stanag_message_received =
  (false -> authin_clk_fall and
  event(authin.stanagout));

assign incoming_stanag_mid = authin.stanag_mid;

assign vehicle_received_stanag =
  loi_clk_fall and
  event(input.loi2vehicle);

assign vehicle_stanag_mid = input.stanag_mid;

assign waypoint_sent_to_fcc =
  input.waypoint_sent_to_fcc and
  input_clk_fall;

assign input_accepts_route = input.route_accepted;

assign fcc_mode = fcc.mode;

assign speed = fcc.speed;

assign altitude = fcc.altitude;

assign stanag_out_receives_ack =
  authout_clk_rise and
  event(authout.stanagin) and
  authout.stanag_mid = 1400;

assign stanag_out_receives_900 =
  authout_clk_rise and
  event(authout.stanagin) and
  authout.stanag_mid = 900;

assign mode = input.mode;
```

Figure 13: The assignment statements used to reference state variables in the top level contract

current LOI is 4. The LOI component guarantees that a navigation message is only forwarded to the `input` component if the LOI is 4.

Figure 16 shows properties that depend only on the state machine described by the guarantees of the `input` component. The state of the mode variable in the `input` component depends on the previous mode and any STANAG 4586 messages that are received. Currently we do not have a complete description of the state machine so we are only able to prove two of the properties. To prove the latter two properties we would need to strengthen the contract of the `input` component to describe in more detail how state transitions occur.

```
guarantee "loi greater than three always gets control" :
  whenever
    received_auth_message_3
  occurs
    acted_on_auth_message
  occurs during [0.0, VSMPkg.system_latency];
```

Figure 14: The guarantee that whenever an authorization message is received and the current LOI is 3 the vehicle accepts this message within a specified latency

```
guarantee "Do not accept NAV commands with loi less than 4":
  vehicle_received_stanag and
  vehicle_stanag_mid >= 800 and
  vehicle_stanag_mid < 1000 =>
  current_loi >= 4;
```

Figure 15: The guarantee that if the vehicle input receives a navigation command then the current LOI is 4

```
guarantee "The aircraft is initially in NO_MODE" :
  (mode = VSMPkg.NO_MODE) -> true;

guarantee "The aircraft never transitions back into NO_MODE" :
  true -> mode != pre(mode) => mode != VSMPkg.NO_MODE;

guarantee "The aircraft can only enter SLAVE2SENSOR mode from WAYPOINT or LOITER mode" :
  true ->
    (mode = VSMPkg.SLAVE2SENSOR_MODE and not pre(mode = VSMPkg.SLAVE2SENSOR_MODE) =>
      pre(mode = VSMPkg.WAYPOINT_MODE) or
      pre(mode = VSMPkg.LOITER_MODE));

guarantee "The aircraft can only enter MANUAL_WAYPOINT mode from WAYPOINT or LOITER mode":
  true ->
    (mode = VSMPkg.MANUAL_WAYPOINT_MODE and not pre(mode = VSMPkg.MANUAL_WAYPOINT_MODE) =>
      pre(mode = VSMPkg.WAYPOINT_MODE) or
      pre(mode = VSMPkg.LOITER_MODE));
```

Figure 16: Guarantees about the state machine in the `input` component.

Possible Counterexamples. There were several properties that we assumed were true about this model, but for which the tool was able to produce counterexamples. The first of these properties is shown in Figure 17. The guarantee states that the vehicle cannot transition into `MANUAL_WAYPOINT_MODE` unless the current LOI is 4 or 5. Intuitively this should be true because in order to transition into `MANUAL_WAYPOINT_MODE` the LOI component must forward a STANAG 4586 message that requires an LOI of at least 4.

However, the tool produces a counterexample where the following scenario occurs:

1. The current LOI is 4 and the `LOI` component receives a STANAG 4586

```
guarantee "The aircraft requires LOI of 4 or 5 in order to transition into MANUAL_WAYPOINT_MODE":
  true -> mode != pre(mode) and mode = VSMPkg.MANUAL_WAYPOINT_MODE =>
    current_loi = 5 or
    current_loi = 4;
```

Figure 17: A guarantee about mode transitions under a certain LOI

message with ID #42. Because the current LOI is 4 this message is forwarded to the `input` component.

2. The `input` component receives this STANAG 4586 message with ID #42 message and begins transitioning its mode.

3. Before the `input` component finishes the `loi` component receives a request to relinquish control (setting LOI to 0).

4. The `input` component completes execution and the vehicle is now transitions to `MANUAL_WAYPOINT_MODE` with LOI 0.

While this counterexample does not seem spurious, it might instead illustrate an error in our formalization of the property. We probably do not care about the value of the LOI the instant that the mode transition occurs. Instead we care that the mode transition occurs in response to a STANAG 4586 message that was received while the LOI was 4 or 5.

The other property that produces a counterexample is shown in Figure 18. The tool produces a counterexample where the following scenario occurs:

1. The `authin` component receives a STANAG 4586 message and forwards it to the `loi` component.

2. The `loi` component has LOI of 4 and forwards the message to the `input` component.

3. The `loi` component forwards a message to the `authout` component. This "erases" the event signal from the `loi` component to the `input` component.

4. The `input` component executes without receiving the route message.

The reason that the tool produces this counterexample is because we do not accurately model the queuing behavior of the real software. In our AGREE model previous messages are overwritten by new messages.

4.2 Resolute

While AGREE is used to reason precisely about how the system behaves over time, Resolute is useful for proving semi-formal properties of a model that do not require reasoning about the system state.

In an ideal world all requirements for embedded control systems would lend themselves to a straightforward formalization. However, it can often be difficult

```
eq route_message_received : bool =
  stanag_message_received and
  (current_loi = 4 or current_loi = 5) and
  (incoming_stanag_mid = 801);

guarantee "A route can be uploaded to the aircraft regardless of state (but correct LOI)":
  whenever
    route_message_received
  occurs
    input_accepts_route
  occurs during [0.0, VSMPkg.system_latency];
```

Figure 18: A guarantee about routes being uploaded to the aircraft

or impossible to express many requirements with formal precision. Resolute is a good tool for capturing and reasoning about these types of requirements. Resolute reasons about the structure of an AADL model. Users express *claims* about how a model is constructed in a formal language. The tool then verifies whether a given model satisfies these claims. If the claims can be satisfied, the tool produces an assurance case whose structure mirrors that of the AADL model and the stated claims. Otherwise, it produces output showing exactly where in the model the claims could not be satisfied.

In the remainder of this section we describe Resolute in detail and give examples of how it was used in the HACMS project.

4.2.1 The Resolute Language

Resolute is a language and tool for constructing assurance cases based on AADL models. Users formulate claims and rules for justifying those claims, which Resolute uses to construct assurance cases. Both the claims and rules are parameterized by variable inputs which are instantiated using elements from the models. This creates a dependence of the assurance case on the AADL model and means that changes to the AADL model can result in changes to the assurance case. This also means that a small set of rules can result in a large assurance case since each rule may be applied multiple times to different parts of the architecture model.

We have implemented Resolute as an AADL annex using the OSATE [8] plug-in for the Eclipse Integrated Development Environment (IDE). Using OSATE, users are able to interact with Resolute in the same environment in which they develop their AADL models. In addition, the resulting framework provides on-the-fly syntactic and semantic validation. For example, references to AADL model elements in the Resolute annex are linked to the actual AADL objects in the same project so that undefined references and type errors are detected instantly.

The syntax of Resolute is inspired by logic programming. Each rule defines the meaning and evidence for a claim. The meaning of a claim is given by a text string in the rule which is parameterized by the arguments of the claim. The body of the rule consists of an expression which describes sufficient evidence to satisfy that claim. Claims may be parameterized by AADL types (e.g., threads, systems, memories, connections, etc.), integers, strings, Booleans, or sets.

4.2.2 Claims and Rules

In Resolute, each claim corresponds to a first-order predicate. For example, a user might represent a claim such as "The memory of process `p` is protected from alteration by other processes" using the predicate `memory_protected(p : process)`. The user specifies rules for `memory_protected` which provide possible ways to justify the underlying claim. Logically, these rules correspond to global assumptions which have the form of an implication with the predicate of interest as the conclusion. For example, an operating system such as Data61's secure microkernel seL4 might enforce memory protection on its own [9]:

```
memory_protected(p : process) <=
  (property_lookup(p, OS) = "seL4")
```

Here we query the architectural model to determine the operating system for the given process. Another way to satisfy memory protection may be to examine all the other processes which share the same underlying memory component. Note that in AADL a "process" represents a logical memory space while a "memory" represents a physical memory space.

```
memory_protected(p : process) <=
  forall (mem : memory). bound(p, mem) =>
    forall (q : process). bound(q, mem) =>
      memory_safe_process(q)
```

In the above rule, we are querying the architectural model via the universal quantification over memory and process components. Note that quantification is always finite since we only quantify over architectural components and other finite sets. The built-in `bound` predicate determines how software maps to hardware. In addition, we call another user defined predicate `memory_safe_process` to determine if a process is memory safe. In the resulting assurance case, the claim that a process `p` is memory protected will be supported by subclaims that all processes in its memory space are memory safe. Thus there will be one supporting subclaim for each process in the memory space.

The above rules for memory protection illustrate a couple of ways to justify the desired claim, but they do not constitute a complete description of memory protection nor a complete listing of sufficient evidence. This is a critical point in Resolute: rules are sufficient, but not complete. The negation of a claim can never be used in an argument (*i.e.*, in logic programming parlance, we do not make a closed world assumption). This is a manifestation of the traditional phrase "absence of evidence is not evidence of absence." Instead, if the user truly wants to use a claim in a negative context, that notion must be formalized as a separate positive claim with its own rules for what constitutes sufficient evidence. For example, one may be interested in a claim such as `memory_violated` which has rules which succeed only when a concrete memory violation is detected.

```
only_receive_decrypt(x : component) <=
  ** "The component " x " only receives messages that pass Decrypt" **
  forall (c : connection).
    (parent(destination(c)) = x) =>
      is_sensor_data(c) or only_receive_decrypt_connection(c)

only_receive_decrypt_connection(c : connection) <=
  ** "The connection " c " only carries messages that pass Decrypt" **
  let src : component = parent(source(c));
  unalterable_connection(c) and (is_decrypt(src) or only_receive_decrypt(src))
```

Figure 19: Example Resolute rules

4.2.3 Computations

Separate from claims, Resolute has a notion of *computations* which are complete and can thus be used in both positive and negative contexts. Usually these computations are based on querying the model. For example, the `bound` predicate above is a built-in computation which returns a Boolean value and is used in a negative context in the rule for `memory_protected`. Users may also introduce their own functions which are defined via a single equation such as

```
  message_delay(p : process) =
    sum({thread_message_delay(t)
           for (t : thread) if bound(t, p)})
```

Here `sum` is a built-in function and `thread_message_delay` is another user-defined function.

Computations may contribute to an assurance case, but they do not appear in it independently since they do not make any explicit claim. Instead, a user may wrap claims around computations as needed, for instance a claim such as "message delay time for `p` is within acceptable bounds" using the `message_delay` function.

Since claims cannot be used negatively while computations can, claims may not appear within computations. This creates two separate levels in Resolute: the logical level on top and the computation level beneath it. The logical level determines the claims, rules, and evidence used in the assurance case argument, while the computation level helps determine which claims are relevant in a particular context and may directly satisfy some claims by performing computations over the model.

External analyses are incorporated in Resolute as computations. An external analysis is run each time the corresponding computation is invoked. This is useful for deploying existing tools for analyzing properties such as schedulability or resource allocation.

Figure 19 shows an example of two Resolute rules. The meaning of the claim is given by the associated text, for example `only_receive_decrypt(x)` means:

```
bound(logical : component, physical : component) : bool =
  memory_bound(logical, physical) or
  connection_bound(logical, physical) or
  processor_bound(logical, physical)

memory_bound(logical : component, physical : component) : bool =
  has_property(logical, Deployment_Properties::Actual_Memory_Binding) and
  member(physical, property(logical, Deployment_Properties::Actual_Memory_Binding))

connection_bound(logical : component, physical : component) : bool =
  has_property(logical, Deployment_Properties::Actual_Connection_Binding) and
  member(physical, property(logical, Deployment_Properties::Actual_Connection_Binding))

processor_bound(logical : component, physical : component) : bool =
  has_property(logical, Deployment_Properties::Actual_Processor_Binding) and
  member(physical, property(logical, Deployment_Properties::Actual_Processor_Binding))
```

Figure 20: Definition of `bound` in the Resolute standard library

"The component `x` only receives commands that pass Decrypt." An instantiated version of this string is what will appear in the corresponding assurance case. The built-in functions like `destination` and `source` return the feature to which a connection is attached, and the built-in `parent` then gives the component which holds that feature. These rules also make use of other user-defined claims such as `is_sensor_data` and `unalterable_connection` which talk about the content and integrity of connections. Note that the two claims shown in the figure are mutually recursive. Together, these claims walk over a model cataloging the data-flow and constructing a corresponding assurance case.

Many claims, rules, and functions will appear within a Resolute annex library which is typically a top-level file in an AADL project. These libraries define the rules for all claims in Resolute, but do not make any assertions about what arguments the claims should hold on. In addition, Resolute comes with a standard library of predefined functions for common operations. For instance, the `bound` predicate for determining if a logical component is bound to a specific physical component is part of the standard library and defined as in Figure 20.

In Phase I of HACMS we prototyped our claims on a simplified AADL model of the SMACCMcopter architecture. This allowed us to test our requirements against a baseline version of the architecture before we started development. A simplified picture of this architecture is shown in Figure 21. Figure 22 shows a portion of a successful assurance case generated by Resolute for this model. Each claim is shown on a single line. Supporting claims are shown indented one level beneath the claim they support. A check next to a claim indicates that it is proven. Figure 23 shows a portion of a failed assurance case. An exclamation point indicates that a claim has failed. In this case, the AADL model includes a safety controller which is allowed to bypass the Decrypt component and directly send messages to the UAV. This bypass is detected Resolute. In fact, the only difference between Figures 22 and 23 is the AADL model. The claims and rules are identical in both.

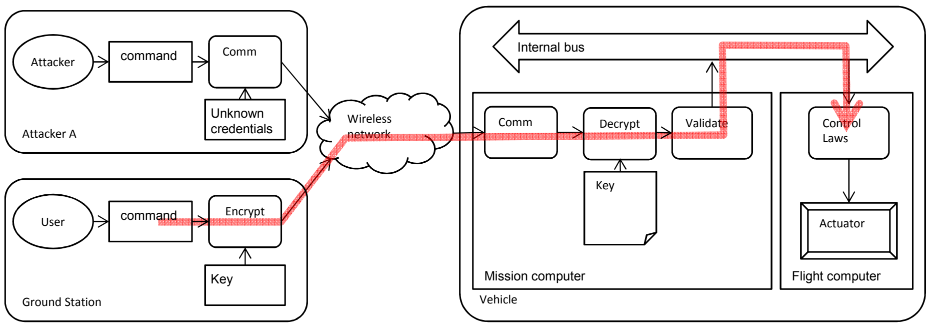
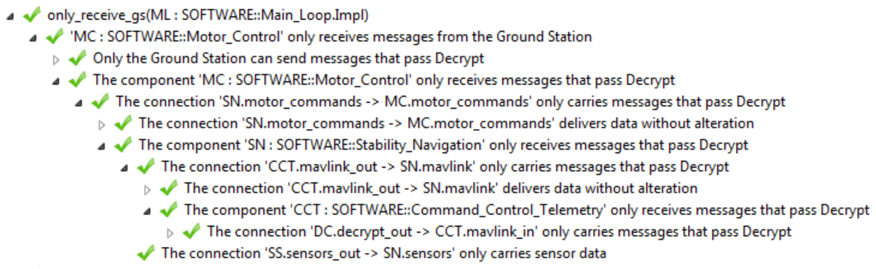
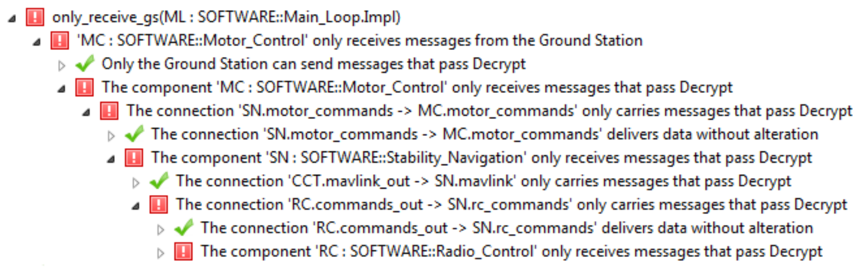


Figure 21: A simplified picture of the software architecture for the SMACCM-copter. The red line illustrates the path that valid commands take to reach the motor controller.



```
◢ ✓ only_receive_gs(ML : SOFTWARE::Main_Loop.Impl)
  ◢ ✓ 'MC : SOFTWARE::Motor_Control' only receives messages from the Ground Station
    ▷ ✓ Only the Ground Station can send messages that pass Decrypt
    ◢ ✓ The component 'MC : SOFTWARE::Motor_Control' only receives messages that pass Decrypt
      ◢ ✓ The connection 'SN.motor_commands -> MC.motor_commands' only carries messages that pass Decrypt
        ▷ ✓ The connection 'SN.motor_commands -> MC.motor_commands' delivers data without alteration
        ◢ ✓ The component 'SN : SOFTWARE::Stability_Navigation' only receives messages that pass Decrypt
          ◢ ✓ The connection 'CCT.mavlink_out -> SN.mavlink' only carries messages that pass Decrypt
            ▷ ✓ The connection 'CCT.mavlink_out -> SN.mavlink' delivers data without alteration
            ◢ ✓ The component 'CCT : SOFTWARE::Command_Control_Telemetry' only receives messages that pass Decrypt
              ▷ ✓ The connection 'DC.decrypt_out -> CCT.mavlink_in' only carries messages that pass Decrypt
            ✓ The connection 'SS.sensors_out -> SN.sensors' only carries sensor data
```

Figure 22: Example of a successful assurance case from Resolute



```
◢ ❗ only_receive_gs(ML : SOFTWARE::Main_Loop.Impl)
  ◢ ❗ 'MC : SOFTWARE::Motor_Control' only receives messages from the Ground Station
    ▷ ✓ Only the Ground Station can send messages that pass Decrypt
    ◢ ❗ The component 'MC : SOFTWARE::Motor_Control' only receives messages that pass Decrypt
      ◢ ❗ The connection 'SN.motor_commands -> MC.motor_commands' only carries messages that pass Decrypt
        ▷ ✓ The connection 'SN.motor_commands -> MC.motor_commands' delivers data without alteration
        ◢ ❗ The component 'SN : SOFTWARE::Stability_Navigation' only receives messages that pass Decrypt
          ▷ ✓ The connection 'CCT.mavlink_out -> SN.mavlink' only carries messages that pass Decrypt
          ◢ ❗ The connection 'RC.commands_out -> SN.rc_commands' only carries messages that pass Decrypt
            ▷ ✓ The connection 'RC.commands_out -> SN.rc_commands' delivers data without alteration
            ▷ ❗ The component 'RC : SOFTWARE::Radio_Control' only receives messages that pass Decrypt
```

Figure 23: Example of a failed assurance case from Resolute

Although the real SMACCMcopter AADL model contained seven times as many software components as the simplified model, very few of the Resolute rules needed to be changed for the assurance case to hold true for the real model. The most significant change was that the true UAV model has data-flow cycles, and therefore the simple recursive rules used in Figure 19 are insufficient. Instead, we created more sophisticated rules which recursively computed the set of components which were reachable prior to passing through the Decrypt component, and then we justified the claim that the given set was complete and did not have access to the motor control component.

Assurance cases as shown in Figures 22 and 23 are interactive in the Resolute user interface. The user can navigate through the assurance case and select a claim to navigate to locations in the model relevant to the claim. For example, the user can navigate to any of the AADL components referenced as input parameters to the claim or can navigate to the rule that defines the claim. This makes it much easier to figure out why an assurance case is failing or why a particular part of the assurance case has a given structure.

An assurance case generated by Resolute is also a stand-alone object. After construction, it no longer depends on Resolute or even the AADL model, though it of course still refers to elements of the model. This means the assurance case can be used as an independent certification artifact. In addition, Resolute allows assurance cases to be exported to other formats and assurance case tools such as CertWare [10].

5 Secure Components

Recent reports of car-hacking via software flaws [11] and insecure low-level networking code [12] point toward the need for safe low-level programming languages. Languages like C or C++ are still the gold standard in embedded system development given the low-level control they provide in terms of memory usage and timing behavior. Unfortunately, these languages provide little support for creating high assurance software—they are unsafe and unanalyzable.

In this section, we describe two embedded domain-specific languages (EDSLs) developed for HACMS, *Ivory* and *Tower*. These languages were ultimately used to re-implement all of the flight control functions in the research vehicle software (referred to as SMACCMpilot) as well as other critical control and communication functions in the ULB.

5.1 Ivory

The language we developed for generating safe embedded C code is called *Ivory*. Ivory compiles to restricted C code suitable for embedded programming. Ivory shares the goal of other "safe-C" languages and compilers like Cyclone [13] and Rust [14]. Our main motivation for not using those languages is our desire for an EDSL providing convenient, Turing-complete, type-safe macro-language (Haskell) to improve our productivity.

There have also been some "safe-C" EDSLs including Atom [15], Copilot [16], and Feldspar [17]. The most significant difference between these languages and Ivory is that they are focused on pure computations (e.g., Feldspar is a DSL for digital signal processing), and do not provide convenient support for defining in-memory data-structures and manipulating memory. Ivory is designed to be a EDSL that can be used for writing safe memory-manipulating embedded C.

Ivory also makes contributions from a programming language perspective, namely in its expressiveness and type-safety. We overview each, then present a small example, to give the reader a feel for the language.

Expressiveness Regarding the expressiveness, Ivory has a variety of useful features, including:

- *Memory-areas*: the ability to allocate stack-based memory and manipulate both local and global memory areas [18].
- *Product types*: C structs with well-typed accessors.
- *FFI*: typed interfaces for calling arbitrary C functions.
- *Bit-fields*: support for typed manipulation of bit-fields and registers [19].

We built Ivory with some limitations to simplify generating safe C programs. Ivory does not support heap-based dynamic memory allocation (but global variables can be defined). C arrays are fixed-length. There is no pointer arithmetic.

Pointers are non-nullable. Union types are not supported. Unsafe casts are not supported: casts must be to a strictly more expressive type (e.g., from an unsigned 8-bit integer to an unsigned 16-bit integer) or a default value must be provided when the cast is not valid. The most common unsafe C cast is not possible: no void-pointer type exists in Ivory.

In Ivory, these have not been limiting factors, particularly because of the power of using Haskell as a macro system. For example, while arrays must be of fixed size at C compile-time, we can define a single *Haskell* function that is polymorphic in the array size that becomes instantiated at a particular size at each use site.

Type-checking Ivory's domain-specific type checking focuses on guaranteeing memory safety and helping programmers reason about their programs' non-functional behaviors more easily.

In addition, Ivory programs have an *effects* type associated with them, implemented as a parameter to the Ivory monad. There are three kinds of effects tracked:

- *Allocation effects*: whether a program performs (stack-based) memory allocation as well as whether pointers point into global or stack memory.
- *Return effects*: whether a program contains a `return` statement.
- *Break effects*: whether a program contains a `break` statement.

Allocation effects allow memory allocation to be restricted and tracked at the type level. For example, from a program's type alone, we can determine whether it allocates memory on the stack, making stack usage easier to track. More importantly for memory-safety, allocation effects also ensure Ivory programs contain no dangling pointers: it is a type error to return a pointer to locally-allocated memory.

Return and break statements fundamentally affect control-flow and can result in unexpected behavior by breaking out of the current block or returning from a function. For example, in a top-level while loop implementing a real-time operating system task, there should be no break or return statements; we can enforce this with the type system. Tracking these effects is novel, we believe, and particularly important in the context of an EDSL in which programs are generated and manipulated heavily in the host language.

In an EDSL, we have at least two options for type checking: (1) write a domain-specific type-checker *in* Haskell (relying on Haskell's type-system just for macro-language type-checking), or (2) embed the domain-specific type checker into Haskell's type system.

We were motivated to pursue option (2) because it allows us to discover problems sooner in the development cycle. In the case of option (1), we only find out about problems in the program's Abstract Syntax Tree (AST) during code generation. Option (2) ensures that all macro and library code is typed correctly, independent of its use in the generated code.

When we began developing Ivory, our hypothesis was that recent type-system extensions to the Glasgow Haskell Compiler (GHC) make it feasible to embed the invariants necessary to ensure memory-safe C programming into the type-system [20]. From a practical standpoint, Ivory demonstrates just how far the type-system has come, allowing us to replicate the type safety of compilers like Cyclone, etc.

We do not have space to adequately describe Ivory's type system; we leave that to a forthcoming paper. Here we will note that the embedding depends on the use of data kinds [21], type families [22], and rank-2 polymorphism [23].

Ivory example We present a small example of Ivory code. The example omits many features of the language, but should give the reader a feeling for it.

Consider Figure 24, in which an Ivory program is shown, as well as the corresponding generated C sources and headers (making a few syntactic changes to the C for readability, not relevant to the example).

First, we define a struct (or product type) using a quasiquoter that is part of the Ivory language. The Ivory code generated by Template Haskell [24] constructs a struct definition containing two fields consisting of an unsigned byte and an array of 10 signed 16-bit integers. Template Haskell also constructs a new type-level literal, `fooStruct`, that is unique to the defined struct. The `Stored` type constructor signifies that the value is allocated in-memory [18]. The `Array` type constructor takes a type-level natural number as a parameter (available as a Glasgow Haskell Compiler extension) to fix the size of an array.

A procedure, corresponding to a C function, has a type of the form

```
Def (params :-> out)
```

where `params` are the procedure's parameter types and `out` is its return type. The procedure `setBaz` takes two arguments and its return type is unit, corresponding to the `void` type in C. The types of the procedure's arguments are types in a type-level list: the first argument is a *reference*, a non-null pointer by construction, to a struct, and the second argument is a signed 16-bit integer. The `Ref` type constructor takes a *scope* type and a memory-area type. The scope type denotes either stack-allocated scope, or global (and statically allocated) scope. In the example, we expect global scope.

Procedures are defined with the `proc` operator that takes a string, corresponding to the name of the function that will be generated in C, and a function from the procedures arguments to its body. The body of the function is an Ivory program that sets each element in the `baz` field of the struct with the value `val` passed to it, leaving the `bar` field unchanged.

Following [18], Ivory guarantees memory-safe array access in the type system since array lengths are statically known. Ivory provides an `arrayMap` operator that applies a function to each valid index into the array. The function applied in this case is a `store` operation that takes a reference to a memory area, a value, and stores the value in the area. It is a type-error if the value's type and memory-area's type do not match.

```
[ivory|
struct fooStruct
  { bar :: Stored Uint8
  ; baz :: Array 10 (Stored Sint16)
  }
|]

setBaz :: Def ([Ref Global (Struct ‘‘fooStruct’’), Sint16] :-> ())
setBaz = proc ‘‘setBaz’’ $ \ref val -> body (prgm ref val)

prgm :: Ref Global (Struct ‘‘fooStruct’’) -> Sint16 -> Ivory eff ()
prgm ref val = arrayMap $ \ix ->
                 store ((ref ~> baz) ! ix) val
```

```
// foo_source.c
#include ‘‘foo_module.h’’

void setBaz(struct fooStruct* n_var0, int16_t n_var1) {
  for ( int32_t n_ix0 = (int32_t) 0
      ; n_ix0 <= (int32_t) 9
      ; n_ix0++ ) {
      n_var0->baz[n_ix0] = n_var1;
  }
}

// foo_module.h
struct fooStruct {
    uint8_t bar;
    int16_t baz[10U];
};

void setBaz(struct fooStruct* n_var0, int16_t n_var1);
```

Figure 24: Example Ivory module definition

The operation (`ref ~> baz`) takes the struct reference and returns a reference to the `baz` field. The bang (`!`) operator takes a reference to an array, an index, and returns a reference to the value at that index. The safety of indexing is maintained since the operator has the type

```
(!) :: Ref s (Array len area) -> Ix len -> Ref s area
```

tying the length of the array to the maximum index. For example, an index type (`Ix 10`) supports index values from 0 to 9.

The example only shows a small part of Ivory's language and does not exhibit some of its additional features to prevent unsafe programs. For example, if `setBaz` had allocated stack memory and created a reference to it, then tried to return the reference (creating a dangling pointer), it would result in a type error.

Additionally, for application-specific properties that cannot be type-checked, Ivory permits the insertion of assertions, assumptions on arguments, and requirements on return values. Ivory also automatically inserts checks for arithmetic underflow/overflow and division-by-zero. All these checks are useful during testing and we have used them to assist with static analysis and model-checking the generated C.

5.2 Tower

In many embedded systems, programmers produce an entire system of software that interacts with multiple input and output peripherals concurrently using a real-time operating system (RTOS). Typical RTOSes provide just a few low-level locking and signaling primitives for scheduling. Since microcontrollers do not have the virtual memory management units (MMUs) found on larger processors, the RTOS kernel cannot protect any system memory against badly behaved user code. These restrictions put significant burden on programmers: they must ensure all tasks, and all communication between tasks, are implemented correctly.

During our initial development of SMACCMpilot, we found ourselves generating high-quality C functions from Ivory, which guarantees memory-safety of the generated code. But whenever we needed "glue code" to implement inter-process communication, initialize data-structures, read the system clock, lock the processor, etc., we were forced to abandon our well-typed world and tediously use C directly via Ivory's foreign function interface. Furthermore, the hand-written C is OS-specific, meaning it would have to be rewritten for any OS port.

Extending Ivory The hand-written glue code was ruining both our productivity and our assurance story. We wanted a language to describe the structure of the glue code that would generate it for us. Our key insight was that such an EDSL could be built as a macro over Ivory, using Ivory's code-generation facilities, without losing anything.

```
blinkTower :: Tower ()
blinkTower = do
  (tx,rx) <- channel
  task ‘‘blink’’ (blinkTask tx)
  task ‘‘lightswitch’’ $
    onChannel rx $
      \lit -> do
        ifte_ lit (turnOn light)
                  (turnOff light)
```

```
blinkTask :: ChannelSource (Stored IBool)
      -> Task ()
blinkTask chan = do
  tx  <- withChannelEmitter chan
  res <- taskLocal
  onPeriod period $ \now -> do
    res <- call blinkFromTime now
    emit_ tx res
  where period = Milliseconds 100
```

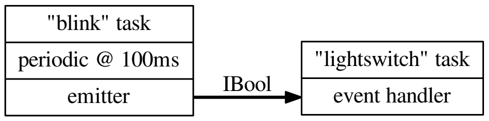


Figure 25: Tower (top), Task (middle), Graphviz output (bottom)

From these ideas, the Tower EDSL was born. Tower is an extension to the Ivory language that is designed to deal with the specific concerns of multi-threaded software architectures. Tower still allows the programmer to use all the low-level power of Ivory for general programming, but uses a separate language for describing tasks and the connections between them. It took about 4 engineer-months and 3k lines of Haskell code to build Tower. This is one of the great productivity features of working with EDSLs: if we discover the language we built is difficult, tricky, or unsafe for solving a particular problem, we can extend that language with a library without modifying the compiler.

In Tower, one specifies tasks and communication channels, and the Tower compiler generates correct Ivory implementations, as well as architecture description artifacts. Tower hides the dangerous low-level scheduling primitives from the user, and keeps type information for channels (i.e., the datatype of the channel message), expressed as Ivory types, in the Haskell type system.

Tower allows the programmer to describe a static graph of channels and tasks. For the intended use case in high assurance systems, a static configuration of channels and tasks simplifies reasoning about memory requirements and permits the system to be analyzed for schedulability.

Multiple interpreters In the Tower front end, the programmer specifies a system that can be compiled to multiple artifacts.

Tower is designed to support different operating systems via a swappable backend. Since all code that touches operating system primitives is generated by Tower, it is easy for the user to specify a system and compile it for different operating systems. Tower supports both the open-source FreeRTOS [25] as well as the formally-verified eChronos RTOS [26] developed by Data61.

Tower also has a backend which generates a system description in AADL [27]. We also built a backend for the Graphviz language to generate graphs of tasks and channels. These output formats make it possible to visualize, analyze, and automatically check properties about the system.

Tower example In Figure 25, we sketch a small Tower example that is representative of a device driver that blinks an LED. Small simplifications to Tower have been made in the code, eliding details relating to code generation and backend selection.

In the first column of the figure, the communication architecture is defined in the `Tower` monad. The program initializes a unidirectional channel between two tasks as well as the tasks themselves. A channel, or queue, consists of transmit (`tx`) and receive (`rx`) endpoints, respectively. The `blinkTask` task is an RTOS task that will send output to the `lightswitch` RTOS task via an RTOS-mediated channel. The `lightswitch` task toggles the LED based on the incoming Boolean values. (In the third column, a graph of the tower program is shown, generated from the Tower compiler's Graphviz dot output, showing the architectural structure of the two tasks as well as the queue between them.)

To conserve space, we only define `blinkTask`. The second column contains the definition of `blinkTask`, defined in the `Task` monad. The `blinkTask` task takes a channel source and returns a task. The task first initializes an *emitter* for the channel then creates a reference to allocated memory that is private to the task. Every 100 milliseconds, an Ivory action is taken. In this case, the action is to call Ivory function `blinkFromTime` that is executed whenever the task is enabled (we elide the implementation of `blinkFromTime` in this example). The boolean value `res` is then emitted on the channel.

6 Secure Operating System

An operating system (OS) is system software that manages computer hardware and software resources and provides common services for computer programs such as scheduling tasks and managing access to memory. From an assurance standpoint, the main objectives of the OS are:

- to provide the software platform and abstractions for EDSL-generated components to run in and communicate with the rest of the system,
- to enforce the high-level software system architecture specified in AADL,
- and to provide the formal base-level guarantees that enable higher-level reasoning and compositional verification.

The OS-level theorems discharge assumptions we make on the model level, as well as individual component assume and guarantee conditions by reducing reasoning to access control configurations. The formal verification of the OS connects both to the architecture-level description of the system, and the verification of components. The SMACCM project relies on two different operating systems: seL4 and eChronos.

seL4 is a microkernel-based operating system that provides isolation guarantees and is used on the mission computers of both the SMACCMcopter and the ULB. The seL4 operating system consists of the seL4 microkernel, the Component Architecture for Microkernel-based Embedded Systems (CAmkES) component platform, and a collection of user-level device drivers and OS services such as network stack, file system and virtual machine monitor (VMM). The seL4 microkernel has formally verified functional correctness, and has formally verified isolation properties, including confidentiality and integrity. CAmkES can generate proofs for architecture correctness and correctness of glue code. For the user-level system components, we use code synthesis and code and proof co-generation to provide assurance of their correctness.

eChronos is a real-time operating system that runs on highly resource-constrained hardware, does not provide isolation guarantees, and is used on the flight computer of the SMACCMcopter. We have verified safe execution, i.e. the absence of undefined behavior, in eChronos' C code by model checking, we have constructed an abstract, but detailed formal specification of eChronos, including its fine-grained interrupt concurrency, and we have proved the correctness of its scheduling mechanism on this model. The fact that eChronos runs on microcontrollers without hardware memory protection means that it is not possible to provide isolation guarantees as in seL4. We therefore use type and memory safe languages (such as Ivory) and static analysis to ensure that user code is well-behaved.

The OS work falls into two broad categories: the OS kernel, such as seL4 and eChronos, and user-level OS components on top of the kernel that deliver services to control components, such as network protocol stacks, file systems, and device drivers.

6.1 seL4 Microkernel

seL4 is a third-generation microkernel that builds on the strengths of the L4 microkernel architecture, such as small size, high performance, and policy freedom, and extends it with a built-in capability model, which provides a mechanism to enforce security guarantees at the operating system and application levels.

seL4's implementation is formally proved correct relative to its specification using mathematical machine-checked proofs in the higher order logic instance of the Isabelle logical framework (Isabelle/HOL) [28]. It is also proved to enforce strong security properties, and its operations have proved safe upper bounds on their worst-case execution times.

As such, seL4 provides the secure software base upon which further secure software layers (system and application services) can be composed to form a trustworthy embedded system.

During the SMACCM project, seL4 was ported to several new platforms (ARM Cortex A15, and NVIDI Tegra K1), extended with formally verified virtualization functionality on the ARM platform (i.e. ARM-hyp), as well as the x64 platform. The formal verification in the project was for functional correctness of the C code in ARM-hyp, and of the executable, design-level specification in x64.

6.2 seL4 Verification

6.2.1 Background: Verified seL4 properties

The past formal verification of the seL4 microkernel includes full functional correctness down to the C implementation level [29], as well as high-level security theorems, in particular integrity and authority confinement [30]. We exploit the functional correctness theorem to provide assurance about the behavior of user-level components interacting with the kernel. We exploit the integrity and authority confinement theorems to discharge architecture-level assumptions on untrusted or less trusted components with very low overhead. The seL4 kernel also comes with a tool set for the automatic initialization of user-level components, in particular initialization of the systems access control configuration. Research conducted independently of this project has delivered a verified version of this tool that we can use to enforce the system-level architecture boundaries described in CAmkES and AADL using kernel-level access control. Further research that was already ongoing on seL4 includes a non-interference theorem that could be used to reason about secrecy, work on extending the formal verification down to the binary level that allows us to remove the compiler and linker from the chain of trusted tools, and extension of the verification to include existing hard real-time features.

The formal verification of seL4 is based on a precise formal semantics of a large, practical subset of C and a reasoning framework for Hoare-Logic and refinement in higher-order logic (Isabelle/HOL), which can be re-used and applied for verifying component implementations, such as control components generated from Ivory, or manually written and verified library components. The corresponding C-to-Isabelle/HOL parser is publicly available under a liberal open source license. The current verification framework is geared towards single-threaded execution. For the verification of the eChronos model, we have extended this framework with Owicki-Gries concurrency [31].

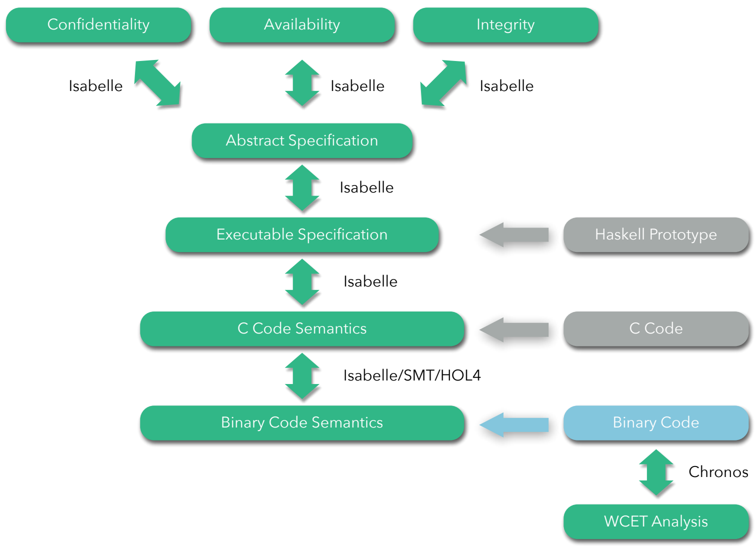


Figure 26: Overview of seL4 verification.

Figure 26 provides an overview of seL4 verification. The key parts are the *abstract specification*, which formally specifies the kernel's API and functionality (i.e. what the kernel does), the *executable specification* which is a formal specification of how the kernel provides the functionality, and the combination of *C code semantics* and *C code*, which models the actual kernel implementation. The functional correctness proof consists of a refinement from the abstract specification to the executable specification, and a refinement of the executable specification to the C code model. Further proofs show that the abstract specification provides *confidentiality*, *availability,* and *integrity* properties. Recent work has further shown that the compiled *binary* is a refinement of the C code model.

6.2.2 ARM-hyp Verification

The ARM Hardware Virtualization Extensions are extensions to the ARMv7 instruction set architecture that provide additional hardware mechanisms for the (more) efficient implementation of hypervisors and virtual machines. In particular, these extensions concern the following main areas:

- a new processor mode, the hypervisor mode (typically shortened to hyp mode). Unlike a similar extension in Intel VT-x to the x86 architecture, this mode is strictly more privileged than the previous kernel mode.[6] The new mode makes it possible to intercept kernel-mode instructions such as setting a new current page table root or cache or translation lookaside buffer (TLB) maintenance instructions. This allows a guest operating system to run unchanged in kernel mode while intercepting and safely emulating these privileged instructions in hypervisor mode. The aim is to remove the need for para-virtualization and to gain the ability to run arbitrary guest operating systems unchanged.

- a new virtual interrupt controller and distributor that makes it easier to hand over interrupt handling to guest OSes directly without emulation. This is mainly a simplification and performance improvement measure.

- a new System MMU, loosely comparable to Intel VT-d extensions that enables a MMU-like translation layer for devices that can perform direct memory access (DMA). It enables the safe isolation of devices used in one guest OS from other guest OSes and from the VMM as well as the underlying microkernel. Unlike the other features above, the System MMU is not implemented on the CPU itself, but on the board the CPU runs on. While the CPU interface is specified by ARM, the implementation and precise feature set of the System MMU is specified by and depends on the board manufacturer.

In general, for microkernel-based hypervisors that support full virtualization, the architecture depicted in Figure 27 has been suggested in the literature.

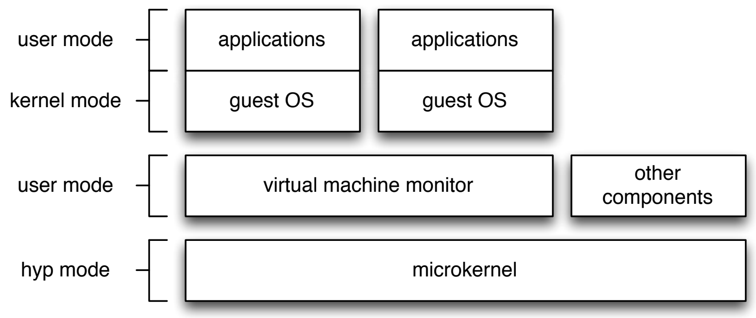


Figure 27: Architecture for microkernel-based hypervisor.

In this figure, the bottom layer is occupied by a microkernel such as seL4, enforcing separation and providing controlled communication as well as further OS mechanisms that the upper layers can use. The second layer is the Virtual Machine Monitor. It uses microkernel mechanisms to emulate a full hardware

[6]We use the terms supervisor mode and kernel mode synonymously in this report

abstraction for guest operating systems running on top. These guest operating systems form the top layer of the architecture. This setup corresponds to a pure hypervisor use case. In a more general setting, as for instance in the later stages of the HACMS program, small trusted native components can be run next to the VMM layer, enjoying isolation guarantees from guest OSes and VMM provided by the microkernel. An additional benefit of separating the hypervisor into a part provided by the microkernel and another part provided by the VMM is that a large part of the VMM can remain untrusted and does not have to be formally verified to provide assurance about other parts of the system.

As mentioned above, the ARM hardware virtualization extensions enrich the set of processor modes with an additional hyp mode of a higher privilege level (PL2) than user mode (PL0) and kernel mode (PL1).

For supporting this mode, the kernel was extended to provide a new assembly-level entry point for hyp mode. The new hyp mode can be entered either by explicit hypervisor call instructions or by highly configurable traps and faults that are generated in kernel mode during guest OS execution.

Since this mode adds new execution state that the kernel must track, we introduce an additional virtual CPU (VCPU) object that stores this additional state and that is associated with the thread control block (TCB) of the guest OS thread. Formally, this adds a new type of kernel object to the specification that models this VCPU object.

In the virtualization setting, threads executing on the microkernel may run in either of two processor modes: kernel mode or user mode. Therefore, the corresponding assembly-level kernel exit code needs to be able to decide which mode to switch to when handing off control. With the new VCPU object associated to a thread, this information is available: if a VCPU thread is associated with a TCB, the corresponding thread runs in kernel mode, otherwise in user mode. On the API and formal specification level this is again mostly visible in the interface to the machine model: the exit transition will lead to the kernel mode or the user mode control state depending on the boolean condition above.

The ARM virtualization extensions add new mechanisms for virtual interrupts and distributing specific interrupts directly to guest operating systems, and allowing the guest to directly acknowledge specific device interrupts.

Since seL4 already provides general-purpose interrupt support to user level components, the existing mechanisms can be used for intercepting and acknowledging device interrupts into and from guest operating systems. We have extended these operations with the ability to inject interrupts into the guest OS.

The virtualization extensions also add new types of faults and traps, i.e. new reasons for entering the kernel, and new types of fault messages to forward from the microkernel to the user and/or VMM.

While the above ARM virtualization extensions are part of the official ARM documentation, the System MMU is not. We have provided a formal specification of ARM System MMU behavior and page table format, but have not included this feature in the verification.

In general the System MMU works by imposing a standard MMU-like address translation to devices with (then not so) direct memory access. Without a System MMU, a device would be configured to read from or write to a specific area of physical memory. Usually this configuration is done via memory-mapped registers, and notification whether new data from the device is available in physical memory happens via interrupts. For the latter two, existing microkernel mechanisms already provide adequate protection. It is the direct access to physical memory that is the problem: a device could be configured by a malicious driver to overwrite kernel memory or other applications. What is more, sufficiently complex devices such as high-performance network cards could be attacked from the outside with the same effect. The main new API visible mechanism is the association between each device and the page table root that defines the address translation for that particular device.

We have proved functional correctness for this new abstract specification and extended C implementation of seL4, including new page tables, new interrupts and interrupt features, new faults types and delivery mechanism, and the new VCPU (guest OS) API mechanisms. As before, the proof is a machine-checked refinement proof between abstract specification and seL4 C code in Isabelle/HOL. It includes new invariants for the abstract specification, an updated design specification, refinement between abstract specification and design specification, as well as refinement between design specification and seL4/arm-hyp C code.

6.2.3 x86_64 Verification

As part of an extension to the original HACMS proposal, we have produced a high-assurance seL4 version for the x64 platform with formally verified, machine-checked functional correctness.

The x64 platform, similarly to ARM-hyp contains hardware extensions for hypervisor support, but in contrast to the ARM-hyp verification had the additional challenge of a completely new architecture on a different basic machine word size (64 instead of 32). The implementation changes resulted in about 50% code change between the ARM and x86 versions of seL4, on 32 bit, and was invasive in implementation, model and proofs, impacting nearly all of the existing artifacts.

As part of this project, we enhanced the tool support for C verification in Isabelle, extending the AutoCorres tool [32] to work incrementally inside an existing proof, increasing verification productivity, and we made extensive use of the new Eisbach [33] proof method language for Isabelle/HOL which we developed in parallel to this project, again to increase verification productivity.

In addition, we invested effort into formally splitting the seL4 proof into generic and architecture-independent parts, using two main mechanisms: Firstly, simple name hiding and qualified name access in the specification artifacts made sure that the generic specification does not make inadvertent use of architecture-specific constants. Secondly, the introduction of formal interfaces between architecture-specific and generic *proofs* in the form of explicit assumptions on the behavior and properties of architecture-specific functions enabled us to reuse proofs about generic functions between the ARM, ARM-hyp, and x64 architectures.

This split required a number of small extensions to the Isabelle/HOL theorem prover to better support access and hiding of qualified names.

The refactoring necessary to implement this split on the abstract and design specifications, as well as the invariant proofs for the abstract specification was significant, but given the demand for multiple further verified architecture ports we consider it a good investment for the future. We intend to carry this formal architecture split further into the access control and information flow proofs, as well as into the refinement proof to further reduce duplication and enhance maintainability.

Due to the limited time scope of this extension, we focused this part of the project on the highest-effort and highest-return part of the proof: functional correctness. Within the time of performance, we achieved the architecture port and split for all specifications, the enhanced tool support for AutoCorres, Eisbach, and Isabelle/HOL, the proof that abstract specification satisfies its correctness invariants (this is the highest-effort part of the overall functional correctness proofs), and the refinement proof between abstract and design specification. At the time of writing, the proof between design specification and C still contains a number of open assumptions that we aim to close after the project.

The x64 port of seL4 contains the following new features:

- **Hardware IO ports.** IO ports provide access to devices. The concept does not exist on the ARM architecture. The x86 kernel protects access to IO ports and provides capability-based access control to them for user programs. For x64, the mechanism is the same as implemented on seL4/x86.
- **IOMMU.** The Input Output Memory Management Unit (IOMMU) on advanced x86/x64 hardware (e.g. most i7 processor boards) controls DMA access for devices. Its main use is for improved performance in virtualization, because it allows a hypervisor to give guests direct hardware device access. The seL4/x64 implementation implements control structures, access control, and initialization for the IOMMU. We have included x64 IOMMUs in the specification, but not yet in the verification of the seL4/x64 port. They are optional (configurable) in the implementation.
- **Virtual Memory.** Virtual memory, i.e. page table data structures and corresponding access control, are different between the ARM, x86, and x64 implementations of seL4. This was the most substantial code part to re-verify.
- **Hardware operations.** The hardware interface for cache and TLB management as well as context switching are different from ARM. While the internals of these functions form assumptions of the proof, i.e. are not verified themselves, their interface use is part of the proof, and therefor led to changes.

- **IRQ handling.** At the beginning of this work stream, seL4's interrupt request (IRQ) handling code was shared between ARM and x86, leading to inefficiencies on x86 and x64. It was redesigned for the verified version, and included in the verification.

6.2.4 WCET and Schedulability Analysis

Knowledge of the worst-case execution times in a critical real-time (RT) system is necessary to determine whether the system is temporally *safe*, meaning guaranteed to meet all its critical deadlines. Given that real-time tasks invoke the kernel for their operation, kernel execution adds to their WCET and must be known for schedulability analysis.

Kernel invocations during execution of real-time code can be explicit, e.g. when sending a message to a different task, or implicit, during interrupts. For (average-case) performance reasons as well as to make verification tractable, the seL4 kernel runs with interrupts disabled. This means a kernel operation cannot normally be preempted, even when the kernel is operating on behalf of a low-priority task while a high-priority interrupt is triggered. This means that the WCET of the kernel is that of the longest-running system call.

By design, seL4 system calls are mostly short, able to execute in a few microseconds at most. However, there are a few long-running operations, mostly to do with unwinding state resulting from derivation chains of access rights and their delegation. In order to limit their latency, the kernel has a number of explicit preemption points. These establish a consistent state of the kernel, including ensuring that of the partially-completed operation, checks for pending interrupts, if there are any, aborts the operation, adjusts its arguments to restart it correctly, and returns to user mode. At that time, interrupts are enabled and the kernel processes them normally. When interrupt handling is completed, the original kernel operation is restarted.

We had in prior work (independent of DARPA funding) developed a tool chain that allowed us to analyze [34] and optimize [35] the WCET of the seL4 microkernel on ARM processors. This work relied on manual (and highly error-prone) determination of safe loop bounds, which are essential to computing a finite WCET bound. Furthermore, standard WCET analysis, which evaluates all code paths through the software, produces many *infeasible paths*, sequences of basic blocks which depend on mutually-exclusive predicates. Eliminating those is important to obtain a reasonable tight WCET bound.

Our previous approach used static analysis of the kernel binary to detect loop bounds and infeasible paths [36, 37]. This had been only partially successful, being able to bound about half the loops in the kernel. The main reason is that the binary is semantics-poor, most of the information of the semantics-rich source code is discarded by the compiler. This is most important for aliasing analysis, i.e. inferring whether two pointers may refer to the same memory location.

In contrast, such analysis is relatively straightforward to do at the source-code level. However, information inferred at the source level does not easily

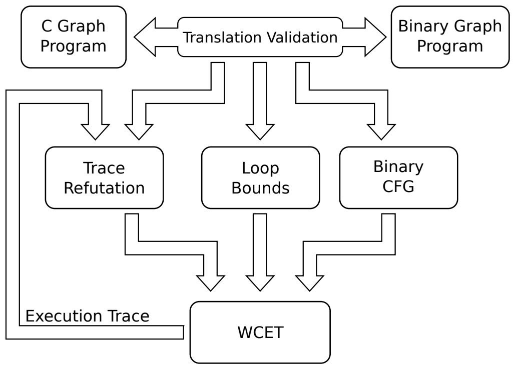


Figure 28: Overview of dataflow in the WCET analysis.

carry through to the binary, as the compiler may re-order and restructure the code.

Under HACMS we took a new approach to solve this problem [38]. Our translation-validation toolchain, which we had in the past used to prove that the seL4 binary was a correct translation of the C source[39], establishes an explicit correspondence between the source and the binary, within our formal verification framework. We can use this correspondence to make assertions about the source code available in the binary, and thus to the WCET analysis.

This allows us to use the wealth of proved kernel properties in the WCET analysis. Furthermore, if the analysis fails to discover a particular loop bound, and the bound can easily be found with some new assertions about the source, we can explicitly insert such an assertion in the source, and make it available to the analysis. Such a new assertion then becomes a new proof obligation in the kernel verification, which is discharged by an explicit proof in the verification framework.

The translation-validation framework provides a high-assurance way of constructing the kernel's control-flow graph (CFG). Constructing the CFG from the binary is difficult and error-prone, extracting it from the validation framework establishes a high-assurance process for producing the CFG. Overall, the WCET analysis not only becomes complete, but also high-assurance.

This new, high-assurance process for WCET analysis is schematically shown in Figure 28. It succeeded in determining upper bounds to *all* loops in the kernel. Furthermore, the small number of extra (verified) assertions needed (22) proved very stable: During one year of significant evolution of the kernel, only one extra assertion had to be added to keep the analysis working correctly. This means that the manual interference through assertions is mostly a one-off effort, after which the analysis can be repeated fully automatically after changes to the kernel.

We also introduced and verified one new preemption point in the kernel, to make a certain, potentially long-running case in object creation pre-emptible.

One serious although presently unavoidable limitation of our analysis is the availability of sufficiently accurate information about the hardware. In the past, ARM published detailed information about the processor pipeline and instruction latencies, allowing us to perform an accurate and sound WCET analysis, e.g. on ARM11 cores (ARMv6 architecture). Starting with the A9 cores (ARM v7 architecture), ARM discontinued this openness, and a sound analysis is no longer possible. Hence we perform all analysis using our sound ARM11 models. The result is not guaranteed to be safe for the more modern cores used in SMACCM. Should a sound analysis be required in the future, this will need cooperation from the hardware manufacturer.

The result of our analysis was to establish a kernel WCET of 1 ms (for a 572 MHz ARM11 processor) for a "closed-system" configuration, where no long-running kernel operations happen after system initialization. All configurations used in HACMS match the closed-system model. The access-control mechanisms of seL4 can be used to enforce the restriction of no long-running system calls.

6.3 CAmkES

CAmkES provides a *component architecture for microkernel-based embedded systems* [40]. Its purpose is to support the development of (embedded) systems on top of microkernels. Since the underlying philosophy of microkernel-based operating systems is to componentize the OS by implementing its services as servers running at user level, it makes sense to apply component-based software engineering techniques to the design and development of these systems. Our approach to doing this involves providing a component platform that supports the modeling of microkernel-based systems as collections of interconnected components. This platform defines a component model, provides languages to describe components and componentized systems, and provides tools that process these descriptions to generate glue code and combine this with hand-crafted component code to produce runnable systems.

The development of CAmkES-based systems has four stages: design, implementation, deployment, and runtime.

At design time a component-based system is defined using an interface definition language (IDL) and an architecture description language (ADL). The IDL is used to define interfaces through which components communicate with one another. The ADL specifies the actual components, including which interfaces they provide and which interfaces they use. The ADL is also used to specify a complete component-based system, that is, the set of components in the system and the connections between the components. Once all the components have been specified, the CAmkES compiler generates header files and stub code from the IDL and ADL.

At the implementation stage the actual component code is written based on the component descriptions and code generated in the previous stage. The implementation consists of regular C code that provides functionality as defined in the header files generated from IDL descriptions. The elements of a system provided by the user, a collection of ADL, IDL and C files, are depicted on the left side of Figure 29 in green.

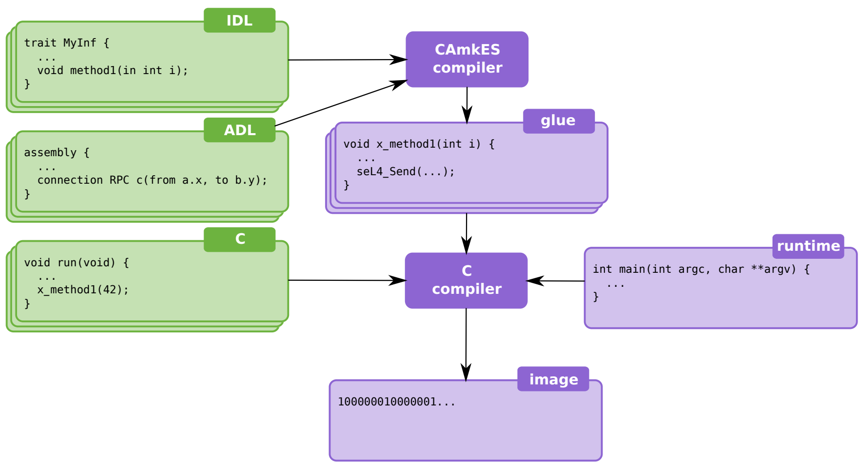


Figure 29: Overview of the CAmkES build process.

At deployment time all of the code (both hand-crafted and generated) is compiled, linked and combined with the core runtime, the operating system and any non-CAmkES services to form a system boot image. Depending on platform requirements this stage may require that extra code responsible for creating, initializing and monitoring components also be generated, compiled and linked into the boot image.

At run time the boot image is loaded onto hardware and the system is started running. During startup, CAmkES runtime code will create and initialize all the components contained in the image. It will also establish any connections between components and initialize any component attributes as defined in the architecture description. After this any component threads are started and the system enters a running state.

The CAmkES platform supports a component model that includes the following architectural elements: components, interfaces, connectors, connections and configurations.

The CAmkES ADL defines components and component-based system descriptions consisting of component instances and connections between them. On the one hand an ADL specification describes the system to be developed, and on the other hand it drives the code generation, compilation, and linking of code required to generate the final runnable system image.

The CAmkES IDL is a subset of the Common Object Request Broker Architecture (CORBA) specification with a few small extensions. Interfaces specified in CAmkES IDL are compiled to a subset of the CORBA C mapping. Us-ing the CORBA nomenclature, IDL files define a number of interfaces, each of which is composed of one or more operations. An operation takes zero or more parameters of various types, and can either return a value or is of type void.

A component is a basic unit of encapsulated behavior. It is used to organize operations and data into interfaces that have well defined semantics and behaviors. Components expose (or export) interfaces that allow applications and other components to access their features. A component must also specify which external interfaces, that is, interfaces provided by other components, it will use.

CAmkES supports three types of interfaces: procedure, event and dataport interfaces. Interfaces are defined separately in the CAmkES IDL.

- Procedure: A procedure interface defines synchronous communication between components through remote procedure calls (RPC). A component must explicitly state whether it provides or uses a procedure interface.

- Event: CAmkES supports a very simple event model. Events are used for asynchronous notifications between components and they are emitted or consumed by components at event interfaces. Events are used to signal other components, but do not carry any data.

- Dataport: The dataport interface represents shared variables that allow components to transfer data between each other. A pair of connected dataports represents the same variable or the same range of memory. Note that this is unlike the data-only interfaces (or ports) defined in other component models that are used to transfer or copy data between components, but do not have sharing semantics. True data-sharing allows us to reduce performance overhead as compared to copying.

The CAmkES component model encapsulates communication between components in explicit architectural elements called connectors and connections (a connection is an instance of a specific connector). A connection is a runtime pathway of interaction between two components. For example, a connector connecting a pair of dataports describes a data sharing relationship between them. In our model, a connector has a name and a list of interface types that it connects. A connector describes a 1-to-1 relationship between interfaces.

In the SMACCM project we extended CAmkES with the ability to generate proofs of functional correctness of the generated code. In order to do this we first developed a formal specification of CAmkES syntax, then defined a formal semantics for CAmkES glue code, and finally added proofs about correctness properties of generated connector glue code [41].

The formal Isabelle/HOL specification of CAmkES defines the abstract syntax of the ADL and IDL in terms of formal datatypes and records. This allows wellformedness constraints to be specified, and consequently for specifications to be checked for wellformedness.

The formal Isabelle/HOL specification of CAmkES semantics defines the behavior of component systems defined by ADL descriptions. This formal specification is intended to apply to the glue code, i.e. the generated communication stubs that are provided to the user by the CAmkES platform. However, the specification goes beyond providing just the glue code semantics. Instead, when combined with specifications of the behavior of user-code, it provides an abstract high-level specification of the behavior of an entire CAmkES-based system.

The specification is high-level, because it abstracts from the underlying kernel mechanisms and message formats. Instead, it is based on a general concurrent message passing framework that can transmit messages of arbitrary high-level types. Instantiating this framework, we restrict it to the kinds of message types of the ADL description and map CAmkES mechanisms to the message passing primitives. Showing that the kernel and glue code indeed implement this high-level semantic view is the main proof obligation of the glue code correctness proof.

The basic communication principle of the underlying semantic framework is synchronous message passing. This is presented in a way that makes it convenient to additionally model atomic asynchronous events and shared memory reads/writes by adding intermediate simulated components. These intermediate model processes map to kernel event buffers and the usual behavior of shared memory pages.

We represent events and shared memory as components. These connector components, unlike the component instances in the system, always have a well-defined, constrained execution because they are effectively implemented by CAmkES and the kernel.

In a static component platform such as CAmkES, correctness can be decomposed into correctness of the components themselves, correctness of CAmkES and correctness of seL4. For the component platform, the property we choose to focus on is the correctness of the generated communication code for RPC communication. More precisely, the desired property is that a remote function invocation (via a CAmkES-provided RPC mechanism) is equivalent to a local invocation of the same function.

With this property available, a user is free to reason about their component-based system as if RPC communication was simply a local function call. Properties that they derive under this abstraction of RPC invocations as function calls remain valid when introducing the complete semantics of CAmkES primitives, as introduced assumptions are discharged by the generated theorems the platform supplies. By composing these generated and hand written proofs together, with the pre-existing kernel correctness guarantees, it is possible to achieve a whole system assurance property while only manually reasoning about the hand written component code.

Similar to how the instantiation of a component system is generated from its ADL description in conjunction with provided user code, we generate the formal specification of a complete CAmkES component system from the same ADL description together with a set of generic base definitions and a set of user-provided behavior definitions for trusted components (i.e. those that are claimed to be more constrained in their behavior than the architecture boundaries enforce).

The definitions of a full system are expected to come from a combination of generated and user-provided theories. The CAmkES generator utility creates

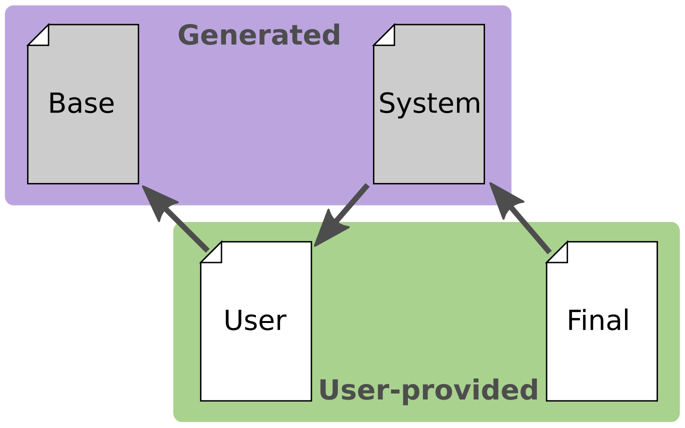


Figure 30: CAmkES theory dependencies.

a base theory using the types and definitions previously discussed that defines primitive operations of a specific system. The user is then expected to provide a theory that defines the trusted components of the system, building on top of these definitions. The generator also produces a theory describing the system as a whole that builds on top of the users intermediate theory. Reasoning about system properties can then be done in another theory building on this generated system theory. These theory dependencies are depicted in Figure 30.

The desirable correctness property of connector glue code is dependent on other, more specific properties. For example, remote procedure call connectors should ensure, among other things, that the function call and parameters that are sent by the caller are correctly received and decoded by the callee. A common requirement for all the glue code is safe execution with respect to the C standard and the state of the system at runtime.

This property requires that the glue code only accesses valid memory, that it obeys the restrictions of the C99 standard and that it always terminates.

In proving this behavior of the glue code, we rely on some explicit assumptions on user code within the system. In particular, we assume that the user code also obeys the C99 standard and does not modify any glue code state. The glue code state covers memory regions relevant for communication with seL4, thread identification and thread-local storage. This state is disjoint from the expected user state; that is, non-malicious user code should never have cause to modify any of the glue code state. As for the seL4 proofs, the generated proofs of CAmkES glue code are intended to apply to an ARM, unicore platform and may not hold in other operating environments.

The current proofs reason about the behavior of the glue code at the level of C, targeting the seL4 microkernel.

It is worth noting that the glue code proofs currently assume valid CAmkES glue code at the other side of the communication mechanism. If the other side of the communication is an untrusted, potentially malicious actor, the proofs do not apply, an extra checking for message wellformedness etc, is necessary. This is relevant for instance for CAmkES components that communicate directly with virtual Linux guest components.

6.4 eChronos RTOS

To accommodate hardware without an MMU in the overall verified setup, we have used a verified RTOS developed by Breakaway and Data61, called eChronos. eChronos is used on the flight control computer in our research quadcopter since that is based on a microcontroller with no MMU. During the HACMS program, we ported eChronos to the ARM and PPC platforms.

eChronos is a real-time OS targeted for use in tightly constrained embedded devices, running on microcontrollers with limited memory and no memory protection. The role of the OS in such systems is closer to that of a library than of a fully-fledged operating environment, allowing the application running on it to be organized in multiple independent tasks and providing a set of API functions that the application tasks can use to synchronize (signals, semaphores, mutexes).

The OS also provides the underlying mechanism for switching from one task to another, and is responsible for sharing the available time between tasks, by scheduling them according to some given OS-specific policy. For instance, tasks can cooperatively yield control to each other (cooperative scheduling); or tasks can be scheduled according to their assigned priority, and their execution must then be preempted if a higher priority task is made available (preemptive scheduling).

The system typically also reacts to external events via interrupts. An interrupt handler needs to be defined for each interrupt by the application. When an interrupt occurs, the hardware ensures that the corresponding interrupt handler is executed (unless the interrupt is disabled/masked).

The job of the scheduler is to ensure that at any given point the running task is the correct one, as defined by the scheduling policy of the system. For instance, in a priority-based preemptive system, when a task in unblocked (e.g. by an interrupt handler sending the signal it was waiting for) a context switch should occur if this task is at a higher priority than the currently running one.

This defines the correctness of the scheduling behavior and is the target of our proof about the eChronos OS.

To reason about such an RTOS, and to prove such a scheduling property, we developed a verification framework supporting the concurrency reasoning required by preemption and interrupt handling on uniprocessor hardware.

We provide a model of interleaving that faithfully represents the interaction between application code, OS code, interrupt handler and scheduler, in such an RTOS [42]. Roughly, the system is modeled as a parallel composition of the code for each application (including calls to OS API functions), the code for each interrupt handler, and the code for the scheduler.

The key feature of the framework is that the interleaving in the parallel composition is controlled using a small formalized API of the hardware mechanisms for taking interrupts, returning from interrupts, masking interrupts, etc. We have formalized our logic in

Using this logic, we have instantiated the model to the eChronos OS and provided a proof of its scheduling behavior [43].

The property we prove is that the eChronos system, starting in any initial state and never terminating, will satisfy the scheduler invariant at every point of execution. The invariant property for the eChronos OS states that the running task is always the highest priority runnable task.

In contrast to the verification of seL4, the verification of the eChronos OS makes assumptions about the safe behavior of user-level code, such as type-safe execution and memory safety to make up for the missing hardware MMU mechanisms. The key for applying such verification successfully is to make discharging these assumptions low overhead, if possible fully automatic. We do this by writing user-code in a type-safe programming language such as Ivory or by performing static analysis on user-code.

6.5 Hardware

During the SMACCM project we had two different hardware platforms for the mission computer. Initially, in phase 2, we used an Odroid-XU, which is a Cortex A15-based computer. However, the Odroid-XU was discontinued, so in phase 3 we migrated to an Nvidia Tegra K1 based computer, the TK1-SOM.

Neither platform provided all the devices that we required (in particular they were missing CAN controllers), so we designed and built daughter-boards containing all the desired peripherals for these platforms. Besides providing peripherals, these daughter-boards also provided power management to allow the Odroid-XU and TK1-SOM to be powered by the SMACCMcopter's battery.

Both daughter-boards were designed with extra sensors and pulse width modulation (PWM) outputs so that they could be used together with their parent computer to function as a flight-controller for the quadcopter (without requiring a separate flight controller computer). This functionality was not used in the SMACCM project, but was added to drive future research on using seL4 in real-time systems.

6.6 Device Drivers

In terms of OS service components, device drivers are the most common source of critical defects. Our approach is to automatically synthesize device drivers from formal specifications.

Termite [44] is a tool, previously developed at NICTA, that performs automatic device driver synthesis. This is a radical approach to creating drivers faster and with fewer defects by generating them automatically based on hardware device specifications. The key idea behind this approach is that the driver synthesis problem can be formalized as a two-player game between the driver and its environments, consisting of the hardware device and the OS. A correct driver implementation can then be obtained from a winning strategy in this game.

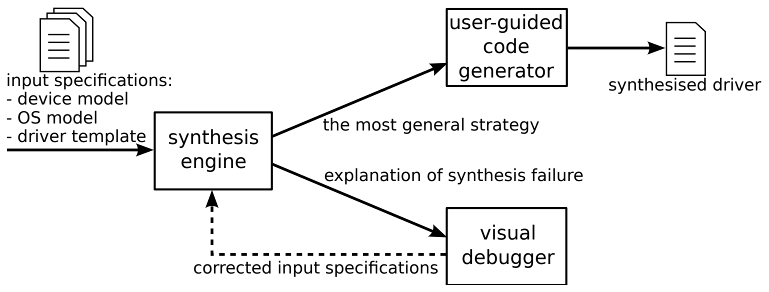

input specifications:
- device model
- OS model
- driver template
synthesis engine
the most general strategy
user-guided code generator
synthesised driver
explanation of synthesis failure
visual debugger
corrected input specifications

Figure 31: Overview of Termite synthesis process.

Figure 31 gives an overview of the driver synthesis process. Termite takes three specifications as its inputs: a device model that simulates software-visible device behavior, an OS model that specifies the software interface between the driver and the OS, and a driver template that contains driver entry point declarations and, optionally, their partial implementation to be completed by Termite.

Given these specifications, driver synthesis proceeds in two steps. The first step is carried out fully automatically by the Termite game-based synthesis engine, which computes the most general strategy for the driver – a data structure that compactly represents all possible correct driver implementations. This step encapsulates the computationally expensive part of synthesis. At the second step, the most general strategy is used by the Termite code generator to construct one specific driver implementation in C with the help of interactive input from the user.

The synthesis engine may establish that, due to a defect in one of the input specifications, there does not exist a specification-compliant driver implementation. In this case, it produces an explanation of the failure, which can be analyzed with the help of the Termite debugger tool in order identify and correct the defect.

The device driver synthesis technology is still in its early days and, as such, has several important limitations. Most notably, Termite does not currently support synthesis or verification of code for managing DMA queues. This code must be written manually and is treated by Termite as an external API invoked by the driver. As another example, in certain situations Termite is unable to produce correct code without user assistance; however it is able to verify the correctness of user-provided code.

For the SMACCM project we have used Termite to synthesize drivers for the Universal Asynchronous Receiver/Transmitter (UART), Serial Peripheral Interface (SPI), and Inter-Integrated Circuit (I2C) devices on the SMACCMcopter

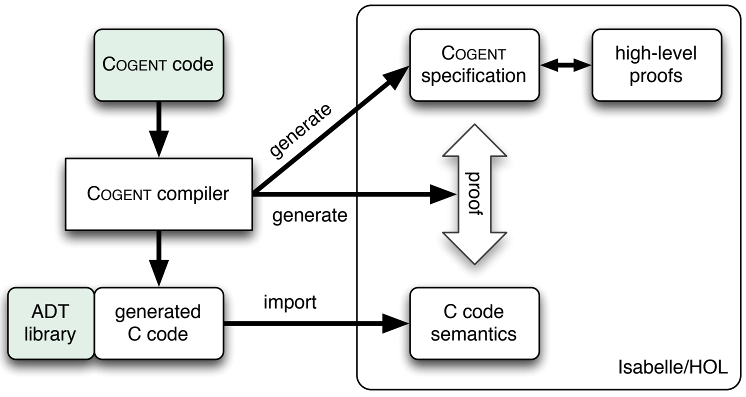


Figure 32: Code/Proof Co-Generation with Cogent.

mission computer.

6.7 File System

A centrally critical OS component for any platform with persistent storage is the file system. When the project started, no file systems that have been verified to the code level existed, even model-level verifications have only managed to prove specific aspects of file systems instead of overall functional correctness. Our approach is to specify the file system code in a higher-level language, and then co-generate C code and functional correctness proofs of the code. We have developed Cogent [45] for this, and used to it to generate several file system components: a custom FLASH file system called BilbyFS, ext2fs and VFAT.

Cogent is a restricted, polymorphic, higher-order and purely functional language with linear types and without the need for a trusted runtime or garbage collector. The Cogent compiler generates a low-level C implementation of the file system, a Cogent specification that we use to verify the functional correctness of the code written in Cogent, and a refinement proof that links the generated C code to the generated Cogent specification. All the specifications and proofs are machine checked in Isabelle/HOL.

Figure 32 depicts the Cogent approach. Cogent compiles to efficient C code, which easily integrates with existing systems APIs and abstract data types, while being designed as a purely functional language, making it very friendly to reason about formally. This makes proving properties of programs written in Cogent far easier than for traditional C code, because the level of abstraction is much higher. Additionally, Cogent's compiler *automatically* proves that the C code it produces is a correct compilation of the functional source program [46]. Therefore, Cogent facilitates obtaining verification guarantees on par with the seL4 approach by mere equational reasoning over pure functions, without unduly sacrificing performance — the remainder of the reasoning is entirely automatic.

A key ingredient for achieving this double-act is Cogent's *linear* type system [47], which allows pure Cogent functions to be compiled to efficient C code without the need for a garbage collector [48] on one hand, while still providing the abstraction necessary to facilitate verification.

The linear type system restricts memory aliasing and avoids, at compile time, many common file system implementation errors such as used-after-free bugs, null pointer dereferences, etc. Cogent's linear type system allows reasoning about purely functional specifications when proving the functional correctness of the code, while generating an efficient C implementation.

Cogent semantics is sequential (allowing asynchronous I/O, but not full concurrency), restricted to total functions, and contains no built-in loops or recursion. This simplifies reasoning, both for the compiler and on top of the language. Iterators, external abstract functions, and types that rely on memory aliasing in C, are invoked via Cogent's foreign function interface and are implemented using a custom template-style C extension. These external functions need to be verified manually using traditional C verification techniques.

We used Cogent to implement three file systems, a custom flash file system called BilbyFs, ext2fs, and VFAT. BilbyFs runs either as a native CAmkES component on seL4, whereas ext2fs and VFAT run as regular device drivers in Linux (either native or virtualized on seL4).

For BilbyFs, we also proved the functional correctness of two key file system operations. We proved functional correctness of these operations against a top-level *abstract* specification of their correct behavior [49] phrased as a nondeterministic functional program in Isabelle/HOL.

Just as with seL4, the functional correctness proofs for BilbyFs relied on establishing global invariants of the abstract specification and its implementation. For BilbyFs, the invariants include e.g. the absence of link cycles, dangling links and the correctness of link counts, as well as the consistency of information that is duplicated in the file system for efficiency.

Importantly, unlike with seL4, none of the invariants had to include the fact that in-memory objects do not overlap, or that object-pointers are correctly aligned and do point to valid objects. All of these details are handled automatically by Cogent's type system and are justified by the C code correctness proofs generated by the Cogent compiler. Thus Cogent considerably raises the level of abstraction for proving properties of systems programs.

Even better, when proving that the file system correctly maintains its invariants, we did so by reasoning over pure, functional embeddings of the Cogent code that the compiler has already automatically proved are correctly implemented by the generated C code.

Our results indicate that Cogent allows verification effort to be meaningfully reduced (by around a third to a half) without unfairly trading away performance.

6.8 CAN Protocol

The Controller Area Network (CAN) protocol is a vehicle bus standard designed to allow microcontrollers and devices to communicate with each other within a vehicle without a host computer.[7] Within the SMACCM project, the CAN bus is used as the communication link between the flight controller and the mission board of SMACCMcopter.

The CAN bus protocol itself is mostly a link-layer protocol that is concerned with bit timing, and priorities of messages if multiple senders access the bus at the same time.

Because of its extreme simplicity at the top level – single packet transmission of 0-8 bytes of data with predetermined unique ID and priority – it is our conclusion that the formal verification of the CAN bus protocol itself is not beneficial for the purposes of the HACMS program. However, there is the need of a fragmentation/reassembly protocol on top of the CAN bus. In this section we will present such a protocol that has been designed and formally modeled by Data61, in cooperation with Galois.

The protocol itself breaks long messages into data-streams that are 8 bytes long. The length of a message (as well as its unique sender and set of possible receivers) is completely determined by its message type; thus a message type determines the number of fragments into which a message will be split – including fragments devoted to authentication and encryption. If a message type prescribes splitting into n fragments, n adjacent CAN identifiers will be allocated to this message type. Thus, a CAN identifier indicates the type of a message, and additionally tells us that it is the kth fragment out of n.

Since split messages can be interleaved by other messages transferred on the same CAN bus, the receiver of the message uses a simple queuing mechanism to sort and store the incoming fragments. A node will use the CAN identifiers to determine if all fragments of a single message have been received. If this is the case the reassembled message is delivered to the corresponding application. In the Open Systems Interconnection (OSI) model, our protocol covers aspects of the layer 3 (network layer) and 4 (transport layer).

The CAN protocol comes with an in-built priority mechanism. It uses a bit-wise comparison method of contention resolution, which requires all nodes on the CAN bus to be synchronized at the point when transmission begins. While this built-in priority mechanism works if only CAN drivers and CAN controllers are considered, it is not sufficient for our research vehicle. In particular it is possible for a priority inversion to occur when a device tries to send a number of low priority messages followed by a high priority message. In this scenario the low priority messages fill a transmission buffer, but cannot be sent due to another device filling the network with medium priority messages. The high priority message would have to wait indefinitely until the medium priority messages stopped and the blocking low priority messages could be sent.

We have developed a protocol, called multiplexer, that addresses this problem and combines the CAN driver and several instances of the fragmentation

[7]http://en.wikipedia.org/wiki/CAN bus

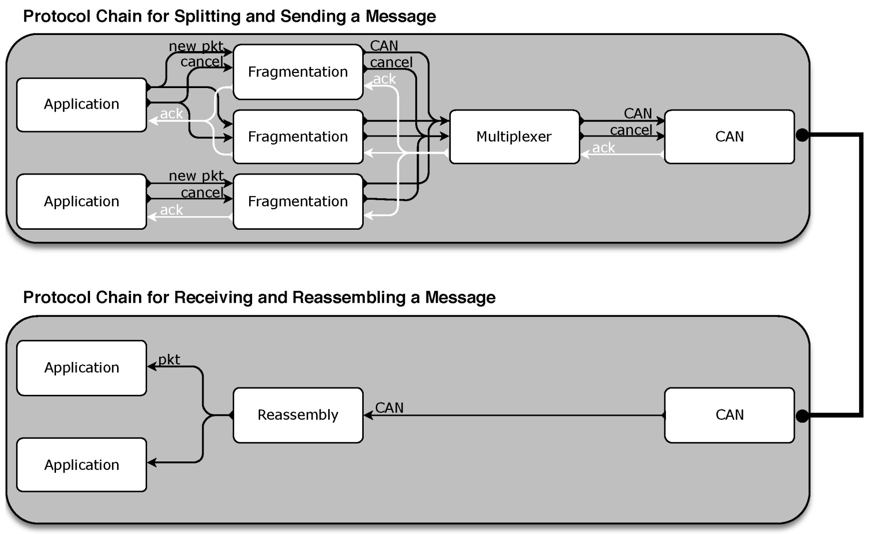


Figure 33: Message passing between the different protocols/components.

protocol.

Figure 33 presents a schematic presentation of our protocol stack. For the purpose of this report we distinguish 4 different layers.

- The application layer: applications are components that send messages to and receive messages from the CAN bus (via the other 3 layers).
- The CAN layer: this layer combines the CAN controller and the CAN driver.
- The fragmentation/reassembly layer: the fragmentation protocol receives a message from an application, and, depending on the type of the message, simply transmits the message via the CAN bus if it is a short or a legacy message, or otherwise splits the messages into fragments of 8 bytes each.
- The multiplexer accepts messages from different instances of the fragmentation protocol (all running on the same hardware, e.g. the mission board) and stores them in a priority queue.

Backwards compatibility with standard CAN should be guaranteed. This is important for legacy nodes that cannot be changed and that can only send standard CAN messages. Such nodes frequently occur e.g. in the automotive industry and so should not be affected by our protocol.

We designed our protocol in a way that ordinary CAN messages can be used and are simply forwarded on and not processed. Such legacy messages have a CAN ID, which doubles as message type, that identifies them as legacy messages; they are not to be split (or equivalently, are split into one fragment). Applications can send legacy messages (or any messages of a type that does not prescribe splitting them) in two ways: (a) the application can just send them to the fragmentation protocol. If the message is registered as one that is not to be split (split into one fragment) the fragmentation protocol will just forward the message to the multiplexer; (b) alternatively, the application could send them straight to the multiplexer.

We have defined a formal specification of the fragmentation and reassembling protocol, the multiplexer, and the CAN driver. Since the specification is fully formal it does not contain any ambiguities.

We have chosen the modeling language Algebra for Wireless Networks (AWN) for the specification. AWN is tailored for modeling and verifying routing and communication protocols and therefore offers primitives such as broadcast. It also defines the protocol in a pseudo-code that is easily readable – the language itself is implementation independent, and it offers some degree of proof automation and proof verification [50], using Isabelle/HOL. AWN is a variant of standard process algebras [51, 52, 53, 54], extended with a local broadcast mechanism and a novel conditional unicast operator—allowing error handling in response to failed communications while abstracting from link layer implementations of the communication handling—and incorporating data structures with assignments. Process algebras such as AWN are equipped with an operational semantics [55, 56]: once a model has been described, its behavior is governed by the transitions allowed by the algebra's semantics. This can significantly reduce the burden of proofs.

To prove properties of the protocol, we decided on a model-checking approach. To make the analysis available for non-experts we have developed a tool that generates input for UPPAAL, one of the most established model checkers. Using AWN and our tool, we believe it is possible that even non-experts in formal methods and verification can perform tasks along the lines of creating, specifying (modeling) and analyzing a protocol. The user need not be an expert, but she must be able to read and write logical formulas; in our case computation tree logic (CTL) formulas. For our verification effort, however, we have to intertwine the automatic reasoning with manual reasoning – in rely-guarantee style. The reason is that we had to deal with state-space explosion.

We proved the following properties of the protocol.

Unreachability of ERROR State In our specification we use a special state ERROR that is reached by a component process whenever it receives an input from another component, or from the application layer, that is unexpected, and for which no proper response in envisioned. The first property of the overall protocol proved is that none of its components will ever reach the ERROR state.

The Protocol is Deadlock Free The next property proved is that the protocol is deadlock free, in the sense that each reachable state has an outgoing transition. As termination of the protocol is not envisioned, a deadlock is a clear case of undesirable behavior. This property does not rule out any state where no further activity occurs due to lack of input from the application layer, for the possibility of such input is modeled as an outgoing transition.

Any Message Received Has Been Sent Here a message counts as sent when it is submitted by the application layer of the sending node to the fragmentation protocol; it counts as received when it is passed on by the reassembling protocol at each node listed as a destination of the message to the corresponding application layer (optionally not counting messages that are discarded immediately by the application layer because they fail to decrypt).

Any Message Sent Is Received This is the central property of the protocol. Obviously, this requirement cannot be guaranteed if there is message loss on the CAN bus. Thus, we must assume that no loss on the CAN bus occurs.

The Application Layer Can Always Succeed in Submitting a New Message The previous property guarantees that, under certain conditions, messages submitted by the application layer to the fragmentation protocol will eventually reach their destinations. As a liveness property for the application layer, this is only convincing if in addition the application layer can always succeed in submitting to the fragmentation protocol any message its want to transmit.

6.9 Ground Control Station Communication Protocol

The protocol used to communicate between the SMACCMcopter and its ground control station is a link-layer protocol that defines how to send packets. Strictly speaking, it is not a complete communication protocol: it defines the type of the messages that are being transmitted, but does not specify a method to ensure reliable receipt and authentication of these messages.

Since the communication is wireless, in order to ensure that the messages received by the air vehicle indeed stem from the ground station, and vice versa, the messages should be encrypted and the sender should be authenticated. Moreover, the protocol needs to ensure reliable delivery of messages, in spite of possible message loss due to unreliable channels or sabotage by an attacker.

We have added functionality to provide secure, authenticated, and reliable communication on top of such basic packet-based protocols.

As recommended by the Red Team, for most of these aspects we looked for, and used, off-the-shelf solutions; there is no need to develop a new protocol if existing protocols meet our requirements. Using existing technology reduces the chance of making errors; moreover, at least some off-the-shelf solutions have been under attack from the outside world and have been shown to be resistant (although formalizations and proofs are often missing).

Our protocol extensions must satisfy the following requirements.

Guaranteed delivery Ideally, we would like a guarantee that each message sent by the ground station is received by the air vehicle and vice versa.

Priority Usually, the ground station and the air vehicle exchange many message. Some of them are important, e.g., a navigation control message directed to the air vehicle. Other messages are of lower priority, e.g., a video stream from the air vehicle to the ground station. The project is working with three priority types on messages: high/medium/low. In cases where not all messages can be delivered due to bandwidth limitations, possibly in combination with other factors, these types determine which messages should be given priority.

Authentication A strict requirement is that the air vehicle only accepts messages from authorized parties such as the ground station. Hence messages should be processed by the air vehicle only if it can be proven beyond a reasonable doubt that they originate from the ground station. We assume that the ground station remains a trusted entity.

Encryption It is an optional requirement, currently deemed less essential than the ones above, that messages between the ground station and the air vehicle cannot be deciphered by intercepting third parties. It is well known that encryption also implies confidentiality.

Integrity Another optional requirement we might want to consider is data integrity. Data integrity refers to maintaining and assuring the accuracy and consistency of data. Note that (symmetric) encryption does not provide integrity. The amount of control that an attacker can have on encrypted data depends on the encryption type. There are off-the-shelf solutions that combine both encryption and authentication. An example is the message authentication code (MAC).

We defined a light-weight encapsulation format that can be used with the communication protocol to protect against forgery, replay attacks, and snooping. The changes result in overhead of 16 bytes of additional bandwidth use per message as well as encryption and decryption operations for each message sent and received.

Our communications security is based on Advanced Encryption Standard using Galois/Counter Mode (AES-GCM). Aside from providing high security, AES-GCM has the advantages of running quickly on most platforms, benefits from hardware acceleration on some platforms of interest, involves no bandwidth increase due to padding, and includes authentication.

6.10 CAN Gateway

The CAN gateway provides a filter between untrusted CAN devices and trusted CAN devices. Trusted devices are typically mission critical and trusted to behave correctly and to not send unauthorized, malformed, or malicious messages.

Untrusted devices, on the other hand, are not considered trustworthy (e.g. they are low-assurance COTS devices) and could send unauthorized, malformed and malicious messages.

It is important to protect the critical trusted devices from the untrusted devices. We do this by placing the trusted devices on a trusted CAN network and allowing no untrusted devices to be directly connected to the trusted network. In this way trusted devices can expect to only receive authorized and correct messages on this network.

However, it must still be possible for untrusted devices to send messages to trusted devices, but we must ensure that only authorized and correct messages are sent from untrusted devices to trusted devices. The CAN gateway's role is to filter and forward messages from untrusted devices onto the trusted network. The filter prevents unexpected, malformed and potentially malicious CAN messages originating from the untrusted device from getting on to the trusted CAN network and being delivered to trusted devices.

There are two options for the design of the CAN gateway.

1. a device that sits between each untrusted device and the trusted network. With this option each untrusted device would have its own CAN gateway.

2. a device that connects the trusted network to an untrusted network. With this option all the untrusted devices would be connected to a single untrusted CAN network and there would be a single CAN gateway device between the untrusted and trusted networks.

We have implemented the second option, however, the design and implementation of the gateway and filter software will be applicable to both options.

Rather than implement the gateway as a separate device, we have integrated it as part of the mission computer, running as an isolated subsystem on seL4. This implementation requires

- a mission computer daughter-board with two CAN controllers so that the mission computer can connect to two separate CAN networks.

- an implementation of the CAN gateway software to inspect, filter, and forward CAN messages

- an implementation of appropriate native seL4 CAN and related device drivers for the mission board, such as SPI and general-purpose input/output (GPIO).

We used the Guardol[57] tool to generate high-assurance code for the filter part of the CAN gateway.

The tool takes as input a set of packet rules, given as regular expressions, that specify the format and content of correct packets. The regular expressions are transformed into deterministic finite automata (DFA) that accept correct packets. The DFAs are then used to generate C code that implements their matching functions.

The code for transforming the regular expressions to DFAs is generated from HOL theorems relating to regular expressions, and is backed by formal proofs of correctness.

The DFA C code is integrated into a CAN filter component that receives CAN packets on one CAN interface, checks whether they have an expected CAN ID, and passes them through the appropriate DFA. If the DFA accepts the packet, then it is sent out over another CAN interface. The CAN filter also implements a simple rate limiter, limiting the rate at which CAN packets will be forwarded.

We developed a demonstration of the gateway functionality, however, it was not part of the final SMACCMcopter. For the CAN gateway demo we filtered on correct GPS CAN messages as defined in `http://www.caemax.de/Downloads/QIC/QIC_GPS_DE.pdf`. From the data sheet: *"Data output takes place continuously with a rate of 1Hz to the CAN bus (IDs 1800 to 1803). Positional data (latitude and longitude) are transmitted separately in degrees, minutes and seconds. Date, time, altitude, speed, and heading data are also available."* We created regular expressions to represent these messages and generated the filter's C code using Guardol. The filter code was combined with CAN drivers and placed into a separate gateway component on the mission board.

7 Trusted Build

Model-driven development is an approach for constructing reliable and secure software systems through the use of engineering models. For cyber-physical systems, architectural models are especially important as they allow representation of both the physical and logical architecture of the system, allowing design exploration and analysis in both of these dimensions. For the past several years, we have been constructing analysis and design tools for AADL that allow a designer to gain confidence in the quality of the architecture and to use compositional formal proof to verify its behavior. Nevertheless, unless the implementation matches the model, this confidence may not carry over into the actual implementation of the system.

One way to ensure conformance is to generate the system image directly from the architectural model. In this section, we discuss our experiences in code generation using AADL. In the HACMS project, we used AADL to specify the architecture and generate all of the glue code for all vehicles used in the final demonstration. The data contained within our architectural model is used as the basis for creating a system image that is loaded directly onto the target platform. Over the course of the project, Trusted Build was used on the following avionics platforms:

- a Pixhawk-based Parrot AR.Drone Quadcopter running CAmkES/seL4,
- a PowerPC-based Boeing Little Bird Flight Control Computer running VxWorks,
- an x86-based Boeing Little Bird Mission Computer running CAmkES/seL4,
- an ODROID-based mission board within an IRIS+ quadcopter
- an NVIDIA Tegra-based mission board within an IRIS+ quadcopter

In addition, Trusted Build was used for the TARDEC Heavy Equipment Transporter (HET) truck and GVR-Bot hardware platforms that ran a combination of CAmkES/seL4 and Linux. The HACMS Red Team was given full access to the design materials and the code of the code generator. Despite the many features, configurations and supported operating systems, no security violations have been found in the generated code to date (the final Red Team report is pending).

This section describes the features of the AADL language as they pertain to code generation, a brief description of the architecture of the tools and generated code, and some information about the build process that was used for HACMS vehicles. A more complete reference for using the Trusted Build tool is found in the *Trusted Build System User's Guide* which is available at the SMACCM github in the `documentation/trusted build` directory.

Although our overall experience with AADL has been positive, we found some deficiencies in the language for modeling certain aspects of our system that were necessary to construct a system image directly from the model. These include:

- Aspects of the code required to generate a system image lacked an adequate representation in the AADL model
- Discrepancies between the thread model used by the target OS and that specified in AADL
- Nonspecific aspects of AADL with respect to event-based communication and scheduling
- Discrepancies between the expected communication model between threads in AADL and our "glue" code
- Managing interactions with "legacy" code that did not fit the communications paradigms used within AADL

We describe these issues and suggest possible improvements to the notation and our processes to ensure that we have very high confidence in system images produced from our architectural models. We would like to both ensure that the system image that we generate is compatible with the analysis tools that we have written for AADL, and that existing AADL analysis tools can understand our (somewhat idiomatic) use of the notation.

7.1 AADL Modeling Language

The architecture for both demonstration vehicles is described using AADL, an architectural description language developed specifically for real-time embedded systems that has been standardized by SAE International [27]. AADL is well-suited to this domain and provides an excellent mechanism for capturing the important details of system design. The AADL can capture both the hardware and software architecture in a hierarchical format. It provides hardware component models including processors, buses, memories, and I/O devices, and software component models including threads, processes, and subprograms. Interfaces for these components and the data flows between components can also be defined. The language offers a high degree of flexibility in terms of architectural and component detail. This flexibility supports incremental development where the architecture is refined to increasing levels of detail and where components can be refined with additional details over time. AADL has both a graphical and textual format, and tools exist for developing models in both of these formats and converting between them. The graphical format, is useful for visualizing the relationship among components while the textual format is the preferred input language for most tools.

Properties are defined for AADL components to specify important configuration data such as execution period, scheduling deadlines, WCET, and bus latencies. This information can be used for schedulability analysis and to support computation of CPU and bus utilization. Data in AADL models provides the basis for generating configuration data for the OS and synthesis of "glue code" that implements component interactions and access to kernel services.

AADL describes the architecture primarily in terms of *components*, which are the hardware or software building blocks of the system, and *connectors*, which describe how they are connected together. Each component has a *component type* specification that defines its external interfaces and attributes. The external interfaces are represented as *features*. *Features* are externally visible characteristics that are used to define the exchange of control and data with other components. Examples of features include ports, bus components, feature groups, and parameter declarations. *Properties* are observable attributes that are represented as typed name/value pairs.

Component interactions are defined through the use of *connections*, which are AADL constructs that link two components and represent the exchange of data and control. There are three types of connections: port connections, parameter connections, and access connections. Port connections represent a directional flow of information between the ports of two concurrently executing components. Parameter connections represent the flow of data between parameters of a sequence of subprogram calls. Access connections designate access to shared components by concurrently executing threads or by subprograms executing within a thread. All of these connection types are supported by Trusted Build and used in one or more of the deployed platforms. .

Port connections may be further classified according to the mechanism of communication used. There are three port categories: *event data ports*, *data ports*, and *event ports*. An event data port represents a message port in which data can be sent and received, and the arrival of data may cause a thread dispatch to occur. If the destination component is busy, the data may be placed in a queue. A data port is essentially an event data port with a queue size of one. By default, a data port does not trigger a thread dispatch. An event port is an event data port with no message and is representative of a discrete event (i.e. a button push).

7.2 Code Generation

The AADL model is the basis for generating operating system configuration data and the C-language source code to be run on the target platform. An overview of the generation process for the eChronos and CAmkES operating systems are shown in Figures 34 and 35. In these figures, the white rounded rectangles represent language and modeling artifacts and the dark blue boxes represent the tools that process these artifacts. The light green boxes represent the output of these processes that, when combined, result in the final system image that is loaded onto the open source vehicle. Although this appears to be a somewhat lengthy and complex process, we have the sequence down to a single command that builds the entire system.

Figure 34 shows the build process for the eChronos operating system, used in the flight control computer of the IRIS+ quadcopter. For this build, Ivory/-Tower generates C code and the AADL model (not shown). The AADL model is parsed by the trusted build plugin, producing an eChronos configuration file and C-language "glue code" that integrates the Ivory-generated code with eChronos.

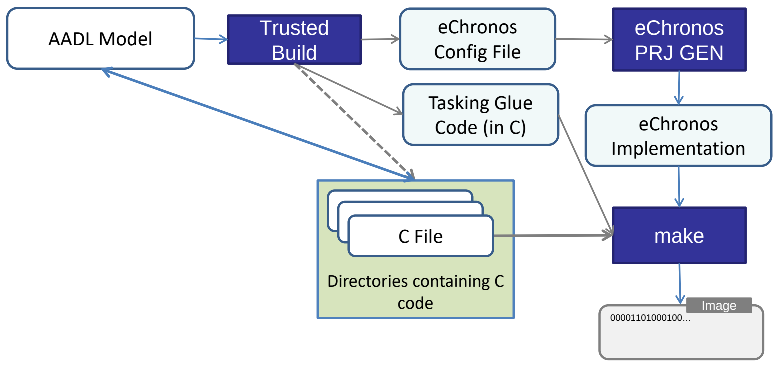
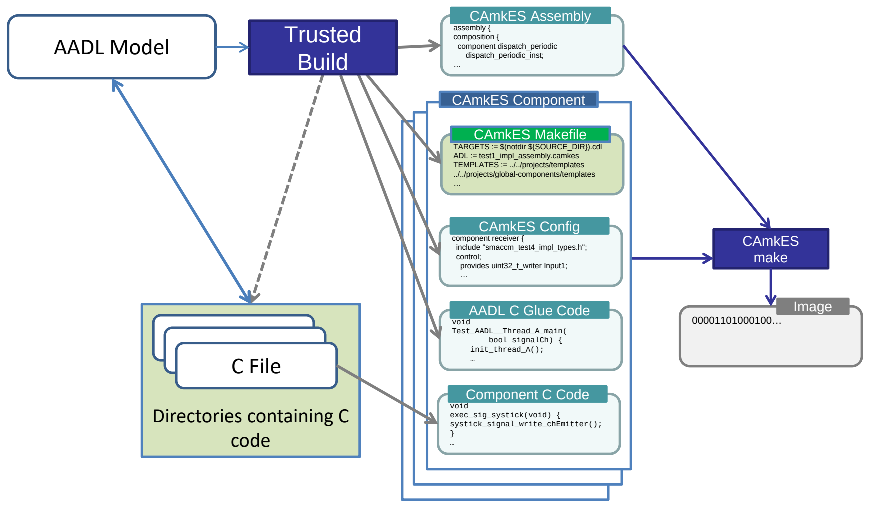


Figure 34: SMACCM Build Process for eChronos



```
assembly {
composition {
  component dispatch_periodic
    dispatch_periodic_inst;
…
```

```
TARGETS := $(notdir ${SOURCE_DIR}).cdl
ADL := test1_impl_assembly.camkes
TEMPLATES := ../../projects/templates
../../projects/global-components/templates
…
```

```
component receiver {
  include "smaccm_test4_impl_types.h";
  control;
    provides uint32_t_writer Input1;
    …
```

```
void
Test_AADL__Thread_A_main(
        bool signalCh) {
    init_thread_A();
    …
```

```
void
exec_sig_systick(void) {
systick_signal_write_chEmitter();
}
…
```

Figure 35: SMACCM Build Process for CAmkES/seL4

The eChronos configuration file is passed to the eChronos build system. After eChronos is built, the glue code and the Ivory-generated code are combined to build the final system image.

Figure 35 shows the build process for the CAmkES/seL4 operating system, used in many of vehicles, including the quadcopter and Little Bird mission computers. For this build, the AADL is created by hand and integrates C code from a variety of sources, including Ivory/Tower. The AADL model is parsed by the trusted build plug-in, producing CAmkES *components* for each thread. CAmkES is a component description language for seL4 and is described in Section 6.3. CAmkES requires make and configuration files for each component as well as an *assembly* file that describes how the components are wired together. CAmkES integrates into make to custom-build the OS and configuration for the application.

To demonstrate the relationship between the model and the generated source, consider the excerpt from the SMACCM AADL model shown in Figure 36. This model contains portions of the definition of two threads (CAN_framing and CAN_driver) as well as a process implementation (Mission_Software.i). The process implementation contains instances of the threads as subcomponents (can_framing, and can_driver, respectively) and also defines connections between the two.

AADL supports definitions of *property sets*, which allow definition of additional properties to the elements of the AADL model. To support system build, we have added a property set called *TB_SYS* that defines information about the components used during build. As discussed in Section 7.1, threads communicate through *ports*. Ultimately, the threads need to invoke API functions that implement the read/write primitives for the port. Normally, the AADL specification would define the names of these communication functions. However, the Galois Ivory/Tower code needs to support both AADL and other middleware products "out of the box." To support this, we allow the user to specify the names and location (in terms of a header file) of the communication primitive. The `TB_SYS::CommPrim_Source_Header` property defines the header file for the thread's functions and the `SMACCM_SYS::CommPrim_Source_Fn` property defines the name of the communication primitive that will be generated from the AADL model. Finally, the `TB_SYS::Is_External` specifies that this thread will be "external", meaning that it does not follow the standard thread dispatch process for trusted build.

The basic model for AADL thread execution is shown in Figure 37, where the code generated by the AADL middleware is shown in blue and the user code is shown in orange gradient. The run() function is the thread entry point generated from the AADL model. In a nutshell, this thread will run forever, waiting for the next event on a dispatch semaphore, which is posted by other threads. Once posted, the thread determines the reason for the post and runs an appropriate *dispatch handler* that determines which actions should be performed. The dispatch handler calls the user entry point, which can then call the middleware communication functions to communicate with other threads.

An excerpt from the corresponding C-code generated from this AADL struc-

```
   thread CAN_Framing
      features
         server2self: in event data port SMACCM_DATA::GIDL;
         self2server: out event data port SMACCM_DATA::GIDL;
         can2self_frame: in event data port SMACCM_DATA::CAN_Frame.i
                  { Queue_Size => 10; };
         can2self_status: in event data port Base_Types::Boolean;
         self2can: out event data port SMACCM_DATA::CAN_Frame.i;

      properties
         Dispatch_Protocol => Periodic;
         Priority => 150;
         Stack_Size => 4 KByte;
         Compute_Execution_Time => 10 us .. 100 us;
         TB_SYS::Sends_Events_To => "{{1 self2can, 1 self2server}}";
         Compute_Entrypoint_Source_Text => "component_entry";
         Initialize_Entrypoint_Source_Text => "component_init";
         Period => 1 ms;
   end CAN_Framing;

   thread CAN_Driver
      features
         framing2self: in event data port SMACCM_DATA::CAN_Frame.i;
         self2framing_status: out event data port Base_Types::Boolean;
         self2framing_frame: out event data port SMACCM_DATA::CAN_Frame.i;
      properties
         TB_SYS::Is_External => true;
         Dispatch_Protocol => Sporadic;
         Compute_Execution_Time => 10 us .. 100 us;
         TB_SYS::Sends_Events_To => "{{self2framing_status,
            self2framing_frame}}";
   end CAN_Driver;
…

   process implementation Mission_Software.i
      subcomponents
         uart_driver: thread UART_Driver;
         decrypt: thread Decrypt;
         encrypt: thread Encrypt;
         server: thread Server;
         can_framing: thread CAN_Framing;
         can_driver: thread CAN_Driver;
         virtual_machine: thread Virtual_Machine;

      connections
         uart2decrypt: port uart_driver.self2decrypt -> decrypt.uart2self;
         encrypt2uart: port encrypt.self2uart -> uart_driver.encrypt2self;
         …
         framing2can_request: port can_framing.self2can ->
            can_driver.framing2self;
         can2framing_status: port can_driver.self2framing_status ->
            can_framing.can2self_status;
         …
   end Mission_Software.i;
```

Figure 36: Excerpt from the SMACCMcopter Mission Software AADL model

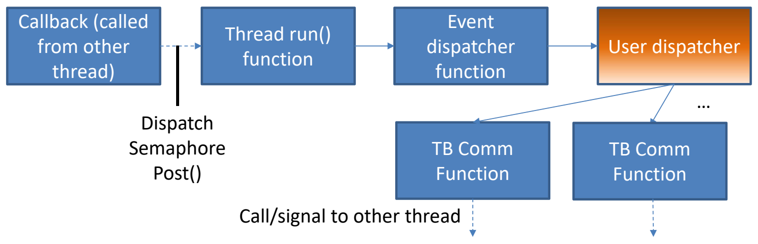

Callback (called from other thread)
Thread run() function
Event dispatcher function
User dispatcher
Dispatch Semaphore Post()
TB Comm Function
TB Comm Function
...
Call/signal to other thread

Figure 37: AADL thread dispatch model

ture is presented in Figure 38. In the case of the CAN framing thread, the only reason for dispatch is the periodic timer, but more complex interactions of dispatchers including event dispatch (due to receipt of an event on an event port) or interrupt service routine (ISR) dispatch (due to processing of a 2nd-level ISR handler). An example of a callback function that "wakes up" the thread is the `periodic_dispatcher_write_int64_t` function. This function is called by the periodic timer; when called, it posts to the dispatch semaphore, which causes the thread to execute. An example comm function is the `tb_CAN_Framing_write_self2can` function, which calls the native CAmkES function to send data to another thread.

7.3 Code Generator Architecture

The trusted build tool is an Eclipse plug-in written in Java. It is designed for extension in two dimensions: support of additional operating systems, and support of multiple implementations of different AADL communications primitives. The "baseline" implementation of port communication primitives (data port, event data port, and event port) uses either local procedure calls with mutexes or RPCs as a mechanism for transmitting data, depending on whether or not the operating system offers memory protection.

Though RPCs work efficiently for communicating data between processes, there is a significant potential security issue. The process acting as the "server" can perform a denial of service on a client simply by preventing the RPC call to complete. The generated middleware code will not allow this to occur, but a malicious actor, once inside the server process boundary, could rewrite this code to prevent return. In addition, if large amounts of data are to be transferred, then RPCs induce a double-copy overhead.

Two alternate implementations of the communications primitives for the seL4 operating system have been created to provide both higher performance and better security.

```
/**********************************************************************
 * periodic_dispatcher_write_int64_t
 * Invoked from remote periodic dispatch thread.
 *
 * This function records the current time and triggers the active thread
 * dispatch from a periodic event. …
 **********************************************************************/
bool periodic_dispatcher_write_int64_t(const int64_t * arg) {
    tb_occurred_periodic_dispatcher = true;
    tb_time_periodic_dispatcher = *arg;
    MUTEXOP(tb_dispatch_sem_post())
    return true;
}
…
/**********************************************************************
 * tb_CAN_Framing_write_self2can:
 * Invoked from user code in the local thread.  This is the function invoked
 * by the local thread to make a call to write to a remote data port.
 …
 **********************************************************************/
bool tb_CAN_Framing_write_self2can
(const SMACCM_DATA__CAN_Frame_i * tb_self2can) {
    bool tb_result = true ;
    tb_result &= tb_self2can_enqueue(tb_self2can);
    return tb_result;
}
…
/**********************************************************************
 * int run(void)
 * Main thread function.
 **********************************************************************/
int run(void) {
    // Port initialization routines
    tb_entrypoint_CAN_Framing_CAN_Framing_initializer(&tb_dummy);

    // Initial lock to await dispatch input.
    MUTEXOP(tb_dispatch_sem_wait())
    for(;;) {
        MUTEXOP(tb_dispatch_sem_wait())

        // Drain the queues
        if (tb_occurred_periodic_dispatcher) {
            tb_occurred_periodic_dispatcher = false;
           tb_entrypoint_CAN_Framing_periodic_dispatcher(
               &tb_time_periodic_dispatcher);
        }
    }
    // Won't ever get here, but form must be followed
    return 0;
}
```

Figure 38: C code generated from SMACCMcopter Mission Software AADL model

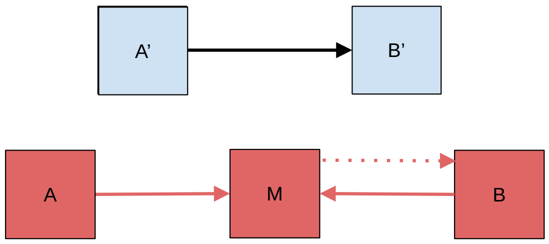


Figure 39: Communicating AADL threads A, B and their implementation in CAmkES

Mailboxes implementing data ports. Other HACMS researchers have implemented a lock-free communication approach for dataports called *mailboxes*. A client publishes data to a mailbox and notifies clients where the most recent data is in a thread-safe way using atomic hardware primitives. The C implementation of mailboxes has been proved correct by Princeton University using the Coq theorem prover. This implementation was used in HACMS ground vehicle demonstrations, but not in the air vehicles.

Monitor components implementing event and data ports. Another approach uses seL4 components as intermediaries between clients and servers for both queued and non-queued communications. Both the client and server RPC into the intermediary monitor component, so a malicious server cannot capture a client.

Consider an AADL model with two threads, A' and B', that share an event port or event data port with output at A' and input at B'. In CAmkES, we will create components A and B to represent A' and B', but also an intermediary monitor component M. A diagrammatic representation of the original AADL model and the generated CAmkES is shown in Figure 39.

We now walk through the interaction between threads through the monitor in Figure 40. We assume that the priority of the CAmkES monitor M is higher than either A and B (or priority inversion can occur). The diagram is fairly self-explanatory, but a few points merit clarification. The dotted line is a seL4 asynchronous notification, so it is not possible for the client to block the monitor. In addition, the enqueue and dequeue methods, used by the server and client, respectively, are *badged* by the operating system so that only the server can use enqueue and the client use dequeue. Finally, CAmkES creates a critical section, so only one or the other method can be active simultaneously (similar to `synchronized` methods in Java), so there is no possibility of race conditions.

In terms of deadlock, there is only one mutex (the sychronizer) and there is no circular wait condition between the monitor and either client, since the notification is asynchronous. What remains to show is that the monitor cannot perform a denial of service on either the client or server. We address this problem by automatically generating a formal proof of correctness for each mon-

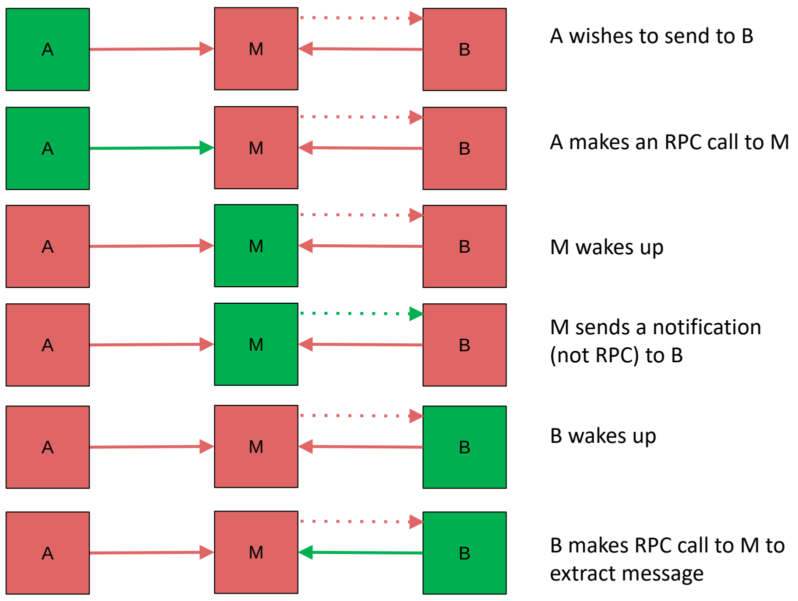

A
M
B
A wishes to send to B
A makes an RPC call to M
M wakes up
M sends a notification (not RPC) to B
B wakes up
B makes RPC call to M to extract message

Figure 40: Flow of interaction between A, B, and monitor M

itor instance. The formal proof relies on the Isabelle/HOL Interactive Theorem Prover and the AutoCorres Isabelle/HOL package.

The proofs rely on an abstraction from the monitor queue from the AutoCorres model of memory to a more traditional list abstract data structure. The reasoning strategy then becomes one of showing enqueue or dequeue on queue abstractions lifted from initial memory result in, after invoking enqueue and dequeue on memory, identical queue abstractions lifted from the modified memory. There are four primary theorems proved within these formal proofs. The simpler of these two pertain to correct behavior of the enqueue and dequeue functions when the monitor queue is full or empty respectively. Specifically, we prove that when the monitor queue is full or empty then enqueue or dequeue invocations, respectively, terminate with a return result indicating failure and alter only memory scoped within the invocations. The more complicated of the two theorems proves that invoking enqueue or dequeue when the monitor is not full or not empty, respectively, terminate with a return result indicating success and extends or shortens the monitor queue, respectively.

8 SMACCMcopter Demonstration

The SMACCMcopter was developed as an open experimentation platform that would be available for use by all researchers in the HACMS program (and even outside of the program). It is based on commercially available hardware components and open source software. It mimicks the architecture and features of the Boeing ULB in a number of ways, and has been a practical way to develop, refine, and test new technologies in the HACMS program.

Like the ULB, the SMACCMcopter includes a flight control computer, a mission computer, an internal data bus, encrypted radio links to a ground control station for command and telemetry, and an onboard camera that can transmit live video data to the ground station. The SMACCMcopter is shown in Figure 41.

8.1 Hardware Architecture

The airframe for the SMACCMcopter is the IRIS+ quadcopter produced by 3D Robotics. The IRIS+ comes with a Pixhawk flight control computer which runs the hard real-time control software and includes integrated sensors for vehicle acceleration and attitude. A separate mission computer has been mounted on top of the IRIS+ body (see Figure 41). The mission computer is composed of a TK1-SOM from Colorado Engineering and a custom I/O daughter-board developed by Data61. It hosts functions for encryption/decryption, ground communication, and the onboard camera. The camera is a PixyCam (CMUcam5) and is mounted on the underside of the IRIS+ body, along with a single-point LIDAR to improve altitude measurement. The mission computer is connected

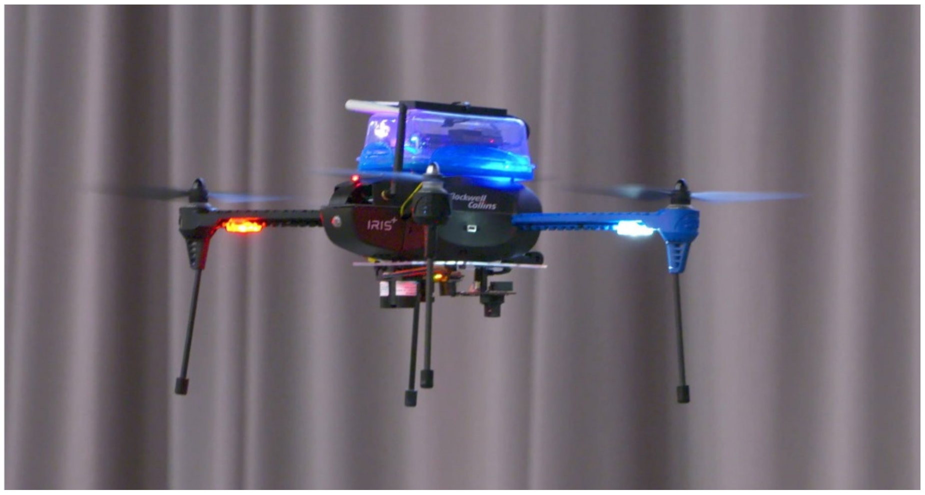


Figure 41: SMACCMcopter during final flight demonstration

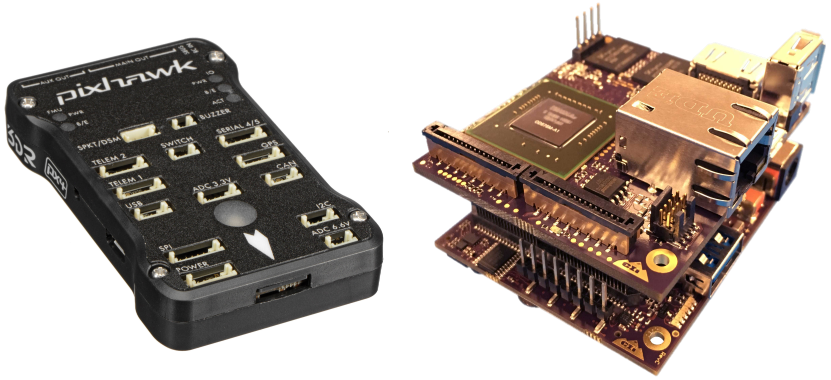


Figure 42: SMACCMcopter computing hardware: Pixhawk flight control computer (left) and TK1-SOM mission computer (right)

to a USB Wi-Fi adapter which is used to stream camera video to the ground station. The camera is also connected to the mission computer via USB. The two computers communicate over a standard CAN bus. The Pixhawk and TK1-SOM (without daughter-board) are shown in Figure 42.

The ground control station software runs on a standard Linux-based laptop computer and communicates command and telemetry data with the SMACCM-copter using a 3DR radio connected via USB. A standard Wi-Fi connection is used for video data. A separate hobby radio controller communicates directly with the flight control computer and provides an independent control mechanism for safety during demonstration flights.

8.2 Software Architecture

All of the SMACCMcopter software was written (or rewritten) from scratch using HACMS technologies.

The software architecture for the flight control computer is shown in Figure 43. The main function of the flight computer is captured by the sensor fusion, stabilization, and motor mixing components. The other components are primarily drivers used to communicate with hardware for the sensors, radio, and CAN bus. Most of the functionality of the original IRIS+ software (which is based on the open source ArduPilot project) was reimplemented in the Ivory and Tower languages, along with an AADL software architecture model. In addition, all of the flight control software runs on the eChronos verified RTOS.

The AADL software architecture model for the mission computer is shown in Figure 44. On the left of the figure, a UART driver receives and sends data over a 915 MHz radio channel to communicate with the ground control station. Separate encryption and decryption components are connected to the UART driver. These components then connect to a central server component which

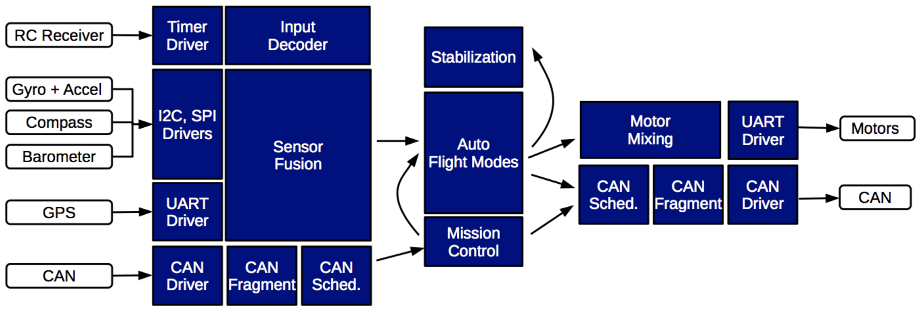


Figure 43: Software architecture for SMACCMcopter flight computer

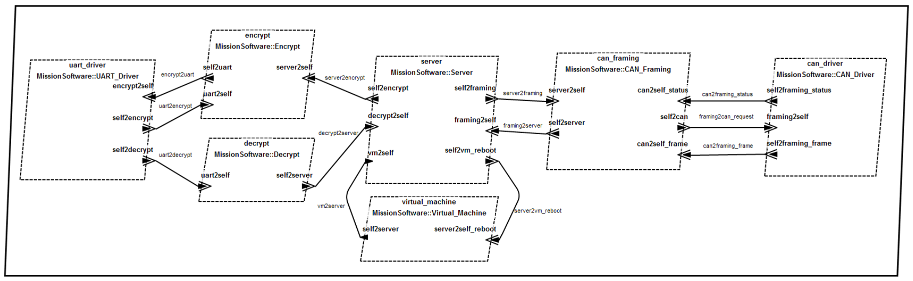


Figure 44: Software architecture for SMACCMcopter mission computer

manages requests from the ground station and responses from the vehicle. The virtual machine component hosts a guest Linux OS and communicates only with the server. The server additionally communicates with the flight computer over the CAN bus using the CAN framing and CAN driver components.

The Linux VM runs the camera software. It receives video data from the camera, detects and computes bounding boxes for objects of a specified color, and sends video data to the ground station over Wi-Fi. The bounding boxes are the only data that are permitted to pass from the Linux VM to the rest of the system.

The central components of the mission computer (Encrypt, Decrypt, Server, CAN framing) are synthesized from Ivory. The other components (UART driver, Virtual machine, CAN driver) are handwritten C code that often communicate directly with underlying hardware.

8.3 Build Process

The build process for both the flight and mission computers is driven by an AADL architecture using the Trusted Build tool. There are variations between the two computers based on how the code for them was developed.

The code and architecture for the flight computer are completely specified

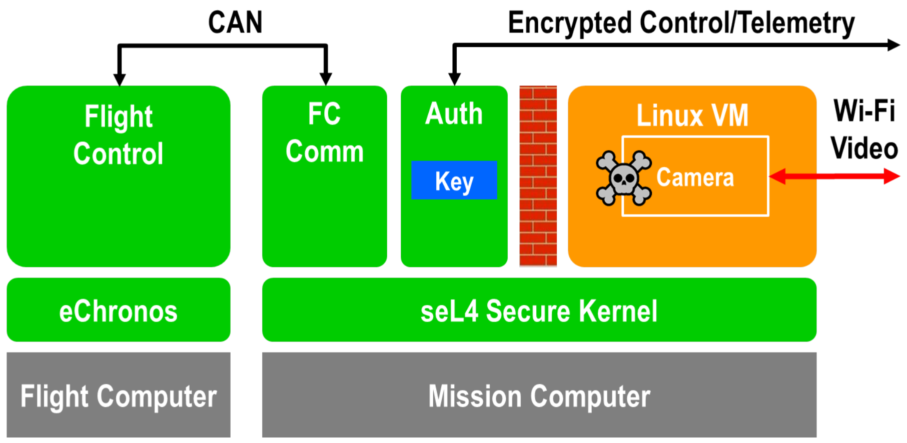


Figure 45: Software architecture of the secure SMACCMcopter, illustrating the attack

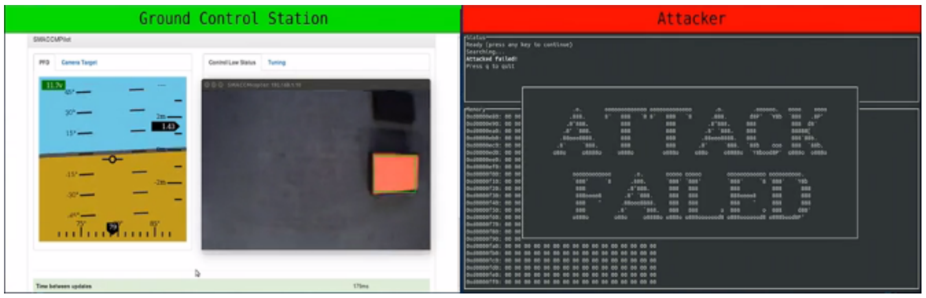
in Ivory and Tower. The build process generates an AADL architecture specification from the Tower description of the architecture. We then use the Trusted Build tool to synthesize configuration files and related glue code for realizing this architecture on top of eChronos. We insert the synthesized Ivory components into this structure, and then build the system image using the eChronos build process. The resulting image can be directly loaded onto the flight computer.

The software architecture for the mission computer is completely specified in handwritten AADL. We use the Trusted Build tool to synthesize a suitable CAmkES configuration with additional glue code for implementing AADL features on top of CAmkES. We insert our synthesized and handwritten components into this structure, and then build the system image using the standard CAmkES build process. This compiles all of our code together with the seL4 kernel and related libraries. The resulting image can be directly loaded onto the mission computer.

8.4 Demonstration

The secure Ivory software components, secure seL4 operating system, and verified AADL software architecture result in a quadcopter design in which most common security vulnerabilities have been eliminated. A simplified diagram of the architecture is shown in Figure 45. The secure components and operating system software are shown in green, while the untrusted camera software shown in orange is isolated in a separate Linux virtual machine (VM).

We showcased the security of the SMACCMcopter in a live demonstration at the Rockwell Collins facility in Sterling, VA in April 2017. The demonstration consisted of two scenarios to shown the difference between an unsecure, unveri-



```
Status
Ready (press any key to continue)
Searching...
Attacked failed!
Press q to quit

ATTACK
FAILED
```

Figure 46: Failed cyber-attack showing ground control station (left) and attacker laptop (right)

fied version of the SMACCMcopter software (that happens to include a security vulnerability) and the secure, verified version of the software. In each scenario, the SMACCMcopter is flown and commanded by the ground control station while a separate team of "attackers" launches a cyber-attack on the vehicle, attempting to take over its telemetry and flight control. In the first scenario, the cyber-attack is shown to be successful, while in the second the SMACCMcopter is shown to resist the same attack and an additional further attack.

In both scenarios, the SMACCMcopter takes off from "friendly territory" and flies over a simulated mission area, streaming back surveillance video via its unsecure Wi-Fi connection. The camera software is hosted in a Linux virtual machine to which the attacker is able to gain access via the Wi-Fi link. From here, the attacker then tries to escalate the attack into a full system compromise.

8.4.1 Successful Attack on Unsecure Vehicle

The first scenario illustrates the possible security consequences of a system with a security vulnerability related to the Linux VM. Exploiting this vulnerability, the attacker is able to access memory outside of the virtual machine address space and find and modify the cryptography keys used for secure communication with the ground station. The attacker overwrites these keys with his own version, causing the legitimate ground station to lose contact with the vehicle, and allowing the attacker to take control of the vehicle. Furthermore, the attacker captures the streaming video while sending a skull image to the legitimate ground station. Finally, the attacker uses the hacked datalink to command the vehicle to leave the mission area and land in "hostile territory."

8.4.2 Failed Attack on Secure Vehicle

The second scenario illustrates the cyber-resilience of the SMACCMcopter running its secure software. Here the attacker is unable to access the cryptography keys from within the Linux VM. Instead, access is denied and only blank memory regions are visible (Figure 46). The attacker then tries to overwhelm the processing capabilities of the system by spawning a forkbomb attack. A forkbomb is a program which continuously replicates itself, thus making it very hard to kill off all instances of the forkbomb. This causes the Linux VM to crash, cutting off attacker access but also crashing the camera software. However, all other aspects of the SMACCMcopter, including telemetry and ground station command, continue to function as usual. The effects of the attack are completely confined to the virtual machine. Moreover, the operator is able to issue a VM reboot command through the secure datalink which restores the Linux VM and the camera software within a matter of seconds. The SMACCMcopter then lands safely in friendly territory.

9 ULB Demonstration

The role of the Military Vehicle Expert in HACMS is to use HACMS technologies to produce a high-assurance version of a defense vehicle. This supports three goals: showing that the technologies can be used on military relevant systems, showing that the technologies can be used by people other than their inventors, and providing an initial assessment of the transitionability of the technologies. Boeing played this role on the SMACCM team. The military vehicle used was the Boeing UH-6 Unmanned Little Bird (Figure 47). The remainder of this section provides additional background on the Unmanned Little Bird, and other activities of the HACMS Military Vehicle Expert, how the SMACCM technologies were applied to the ULB, two flight demonstrations that validated the application of the HACMS technologies to the ULB, and the results of formal analysis of security properties of the HACMS ULB. This section concludes with lessons learned and suggestions for future evolution of the HACMS technologies.

9.1 Background

9.1.1 Challenge Problems

As part of its role as military vehicle experts, Boeing provided multiple releases of sets of challenge problems that were intended to represent particular situations, contexts, or areas where HACMS technologies could be employed

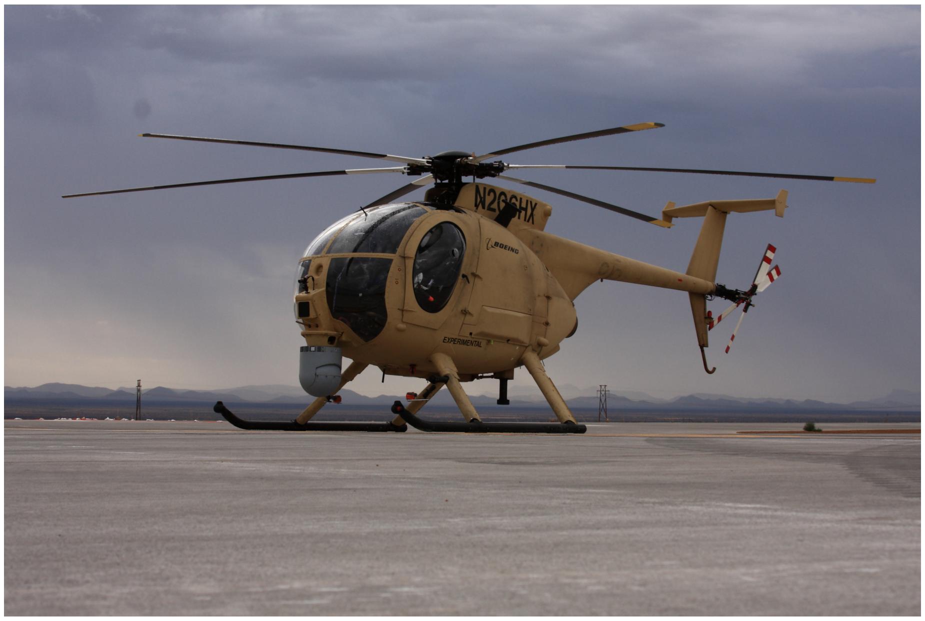


Figure 47: The Boeing Unmanned Little Bird

to improve the process of developing secure embedded vehicle systems. These benefits can include improving the assurance of security of the resulting system and improving the effectiveness of the process of developing secure systems. The challenge problems were intended to both guide HACMS research based on the needs of militarily relevant embedded vehicle systems, and to provide a mechanism for demonstrating the resulting technologies.

The challenge problems provided basic information about the ULB, including a shareable system level AADL model, and pseudo code for determining the authorization of ground stations to issue commands to the aircraft based on their designated Level of Interoperability. With this information, generic and ULB specific challenge problems were defined. Generic challenge problems were applicable not just to the ULB or military air vehicles, but to military vehicle systems in general. Generic challenge problems included ensuring that only authorized commands are executed by a vehicle, or that maintenance is performed securely. ULB specific challenge problems included enforcing ground station LOI restrictions, and ground station authentication.

9.1.2 Unmanned Little Bird

The ULB is an optionally manned rotary UAV, based on the H-6, a 32 foot long, 4700 pound rotorcraft. The ULB adds an autonomous capability to the basic H-6. This is provided by a Vehicle Management System (VMS) containing a Flight Control Computer and a Vehicle Specific Module. Though the ULB is capable of fully autonomous flight, it is generally flown with a safety pilot, who can disable and override the ULB VMS. The presence of the safety pilot and the ability of the safety pilot to override the VSM ensures safety of flight regardless of VSM changes, and permits the ULB to fly under FAA Optionally Piloted Experimental Aircraft rules, rather than under Unmanned Air Vehicle rules. These two factors make the ULB an attractive research vehicle for programs like HACMS. As a result, the ULB program has supported many UAV technology development programs since its first flight in September 2004. The ULB system is comprised of three basic subsystems, the Ground Control Station (GCS), the data link, and the air vehicle (Figure 48).

The GCS is comprised of three main components interconnected via Ethernet. The primary piece is the Common Unmanned Control System. The CUCS is a STANAG 4586 compliant system for control of multiple dissimilar UAVs. The CUCS used by the ULB is Boeing's COMC2 ground control station. COMC2 communicates to the aircraft using User Datagram Protocol (UDP) Multicast via the data link. A security gateway router is used when the GCS needs to connect to external systems.

The second subsystem is the data link. The ULB data link is the L-3 Com Mini-TCDL, a Ku-Band, encryptable, wireless Ethernet data link. The Ground Data Terminal (GDT) is the ground portion of the data link. The GDT consists of a Mini-TCDL data modem, radio frequency equipment (RFE), and 2-Axis directional antenna. The Airborne Data Terminal (ADT) also consists of a data modem and RFE, but uses an omni-directional antenna.

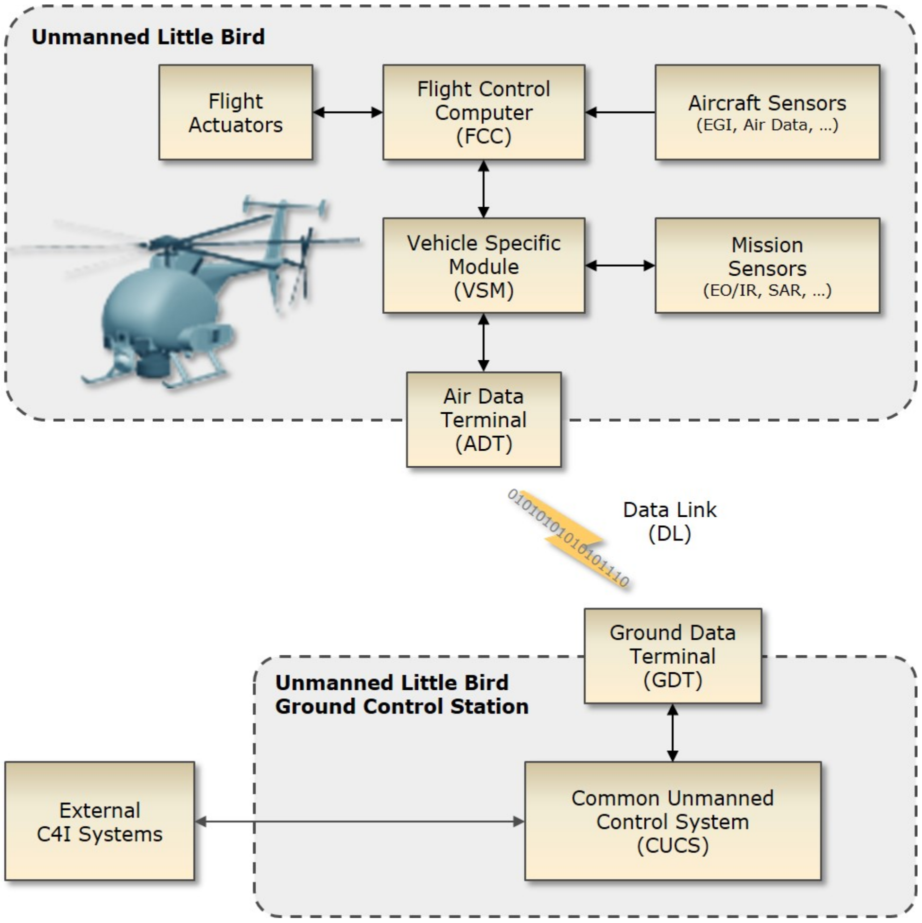

Unmanned Little Bird
Flight Actuators
Flight Control Computer (FCC)
Aircraft Sensors (EGI, Air Data, ...)
Vehicle Specific Module (VSM)
Mission Sensors (EO/IR, SAR, ...)
Air Data Terminal (ADT)
0101010101010101110
Data Link (DL)
Ground Data Terminal (GDT)
Unmanned Little Bird Ground Control Station
External C4I Systems
Common Unmanned Control System (CUCS)

Figure 48: The Unmanned Little Bird Unmanned Aircraft System

The air vehicle subsystem includes a single chassis VMS which contains two main processing units: the VSM and the FCC. The VSM is designed to implement the STANAG 4586 [58] UAV control standard. The VSM essentially acts as a translator converting STANAG messages into the appropriate internal ULB proprietary messages. The VSM has the secondary task of managing mission sensors such as the Wescam MX-15 EO/IR sensor. The FCC manages the flight control system for the aircraft and all of the flight critical aircraft sensors such as the Enhanced GPS and Inertial (EGI) navigation system, Radar Altimeter, and Air Data Computer. The VSM communicates with the GCS via UDP multicast; to the Wescam via UDP; and to the FCC via UDP. The FCC communicates to the flight control actuators and aircraft sensors using a variety of interfaces including digital and analog I/O, RS-232, ARINC-429, and MIL-STD-1553B. The FCC communicates with VSM via Mil-STD-1553B in the original ULB configuration. During the HACMS program, the ULB program replaced this with a dedicated Ethernet connection. A small helper application on the VSM processor board acts as a 1553 to Ethernet bridge.

The ULB implements the STANAG 4586 protocol for communication between ground stations and UAVs. The protocol was designed to permit any compliant ground station to control any compliant UAV. Though developed by NATO, the third edition of the protocol has been publicly released. The protocol defines roles for the command station and the aircraft vehicle specific module. Though the protocol allows for the VSM to be on the ground with communication to the aircraft via an aircraft specific protocol, the ULB VSM is actually on the aircraft, so that the air-ground link uses the STANAG 4586 protocol. In addition to defining messages for waypoint navigation and payload control, the STANAG 4586 protocol defines 5 Levels of Interoperability, which specify the types of relationships between a ground station and an aircraft.

- LOI 1:Indirect receipt of payload data.
- LOI 2: Direct receipt of UAV related data.
- LOI 3: Control and monitoring of the UAV payloads
- LOI 4: Navigational control and monitoring of the aircraft less launch and recovery
- LOI 5: Navigational control and monitoring of the aircraft including launch and recovery.

For a given UAV and payload, there can be any number of ground stations with LOI 1 or 2, at most one with LOI 3, and there should be exactly one with LOI 4 and LOI 5 combined. During the baseline assessment it was noted that the number of ground stations at LOI 2 needs to be bounded, otherwise a denial of service attack could occur, with all UAV bandwidth devoted to publishing data to an arbitrarily large number of ground stations.

The ULB offers four autonomous navigation modes. In waypoint mode, the aircraft follows routes defined by waypoints, which include altitude and velocity.

In loiter mode the ULB orbits a designated loiter point. In slave to sensor mode, the ULB loiters around a waypoint defined by the stare point of the onboard camera. Finally, there is an autopilot mode where the ground station directly supplies heading, airspeed, and altitude.

The original ULB VSM is implemented in C++ in a component/library style. It consists of approximately 87 KSLOC of source code, with an executable size of approximately 80 MB. It runs on Gentoo Linux on the aircraft, and is built and tested on Kubuntu Linux. It is hosted on an x86 platform. The baseline aircraft used a Dynatem Pentium-M Processor, but during the course of the HACMS program this was replaced by an AiTech, 1.33 GHz, C162 processor board with an Intel Core i7 processor, since the original processor did not have the virtualization extensions required by seL4.

The original ULB FCC was written in C using a monolithic cyclic executive running on a board support package at 50 Hz. It consisted of approximately 20 KSLOC of source code, with an executable size of approximately 2 MB. During the HACMS program the Boeing ULB program ported the FCC software to VxWorks, which increased the size to approximately 40 KSLOC. The baseline aircraft used a DY 4 DMV-181-2828 Power PC Processor Module, but due to parts obsolescence issues, this was replaced by a Curtiss-Wright DMV-194.

9.2 Applying HACMS Technologies to the ULB

Over the course of the three phases of the HACMS program, HACMS technologies were progressively applied to the ULB to create a high-assurance cyber military system. This section will describe how the different HACMS technologies were applied to the ULB. In HACMS Phase 1 and 2, SMACCM technologies were applied to the ULB VSM, while in Phase 3, HACMS technologies were applied to the FCC as well. Figure 49 shows the baseline ULB architecture at the start of the HACMS program. In HACMS Phase 1, the system architecture was modeled in AADL, and seL4 was used as a hypervisor to host the baseline VMS on its baseline Linux operating system as a guest operating system (Figure 50). In the figure red represents non-secured components, purple represents partial application of SMACCM technology, and blue represents full application of SMACCM technology.

During Phase 2 AADL was used to represent the VSM software architecture sufficient to enable use of the SMACCM AADL2RTOS tool to generate glue code for the VSM, Ivory was used to implement a subset of the VSM, and the authentication and LOI components. The remaining elements of the VSM were left as legacy components executing in Linux virtual machines. All of the VSM software then ran on top of seL4 (Figure 51). The resulting system was used in a flight demonstration on 24 July 2015 to show that improving security does not preclude satisfying the Cyber-Physical Systems (CPS) requirements of flying systems.

While the VSM is an embedded system and has some real-time requirements, it is the FCC and the flight control algorithms implemented there that have the most stringent hard real-time CPS requirements. In Phase 3, the FCC

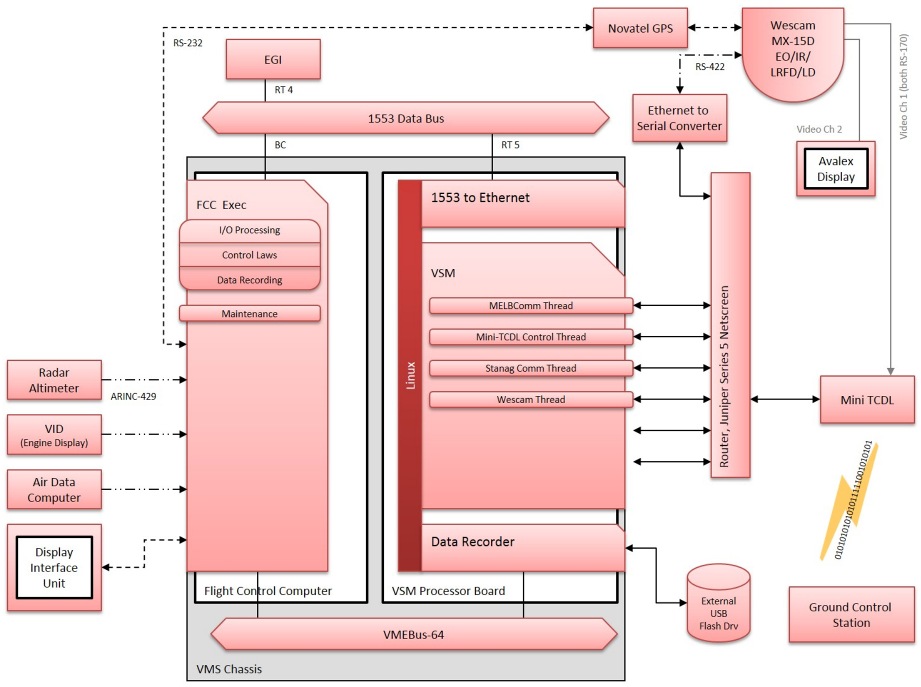


Figure 49: ULB Baseline Architecture

software architecture was modeled using AADL, and the outer loop control and input/output elements of the FCC were implemented in Ivory. In this case the existing VxWorks RTOS was retained as the operating system. The resulting final ULB HACMS architecture is shown in Figure 52.

9.2.1 Architecture

The ULB system and software architecture was modeled using AADL. The AADL model served three roles. First, the AADL model was used to capture the overall design of the system, and track the evolution of the system as HACMS technologies were applied to the system. Second, the AADL model was the basis for specification and verification of security properties (see Section 4.1.1). SMACCM adopted an architectural approach to reasoning about system security. This is appropriate since security is a system level property. Finally, the system AADL model was used for code generation. Component interfaces and interconnections were defined in the AADL model, and then glue code implementing those interfaces and interconnections was generated from the model.

An initial open system level model was developed during phase 1 that captured the hardware aspects of the ULB avionics, with a simple software model. Then, over the course of the program as the VSM and later FCC were reimple-

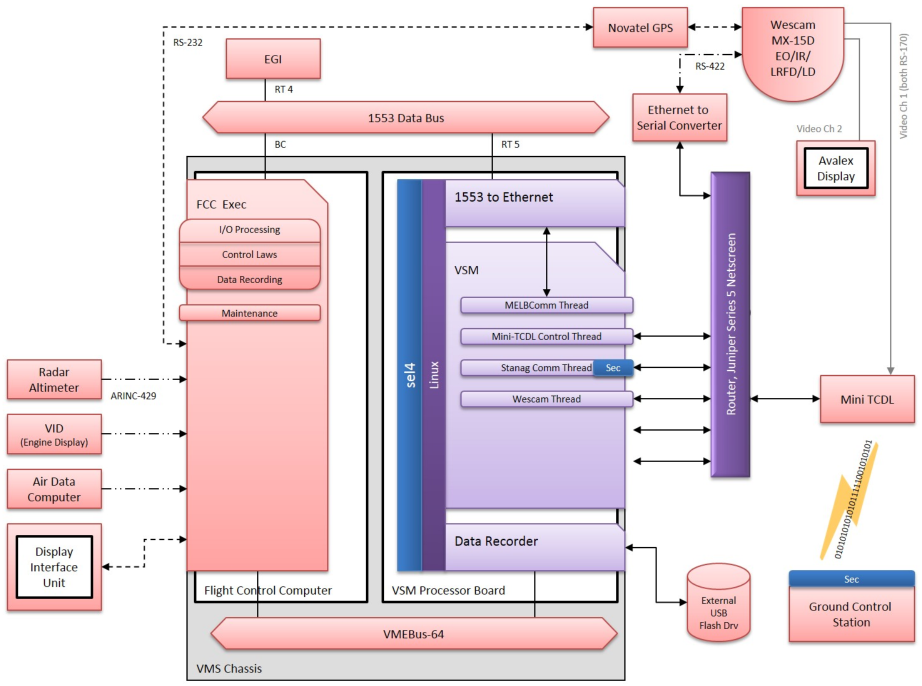


Figure 50: ULB HACMS Phase 1 Architecture

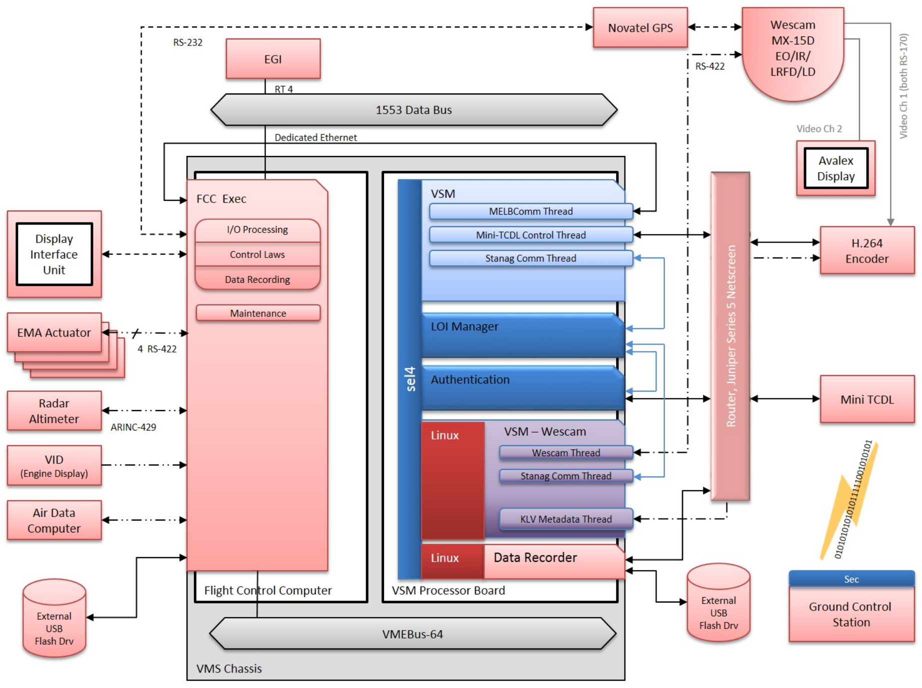


Figure 51: ULB HACMS Phase 2 Architecture

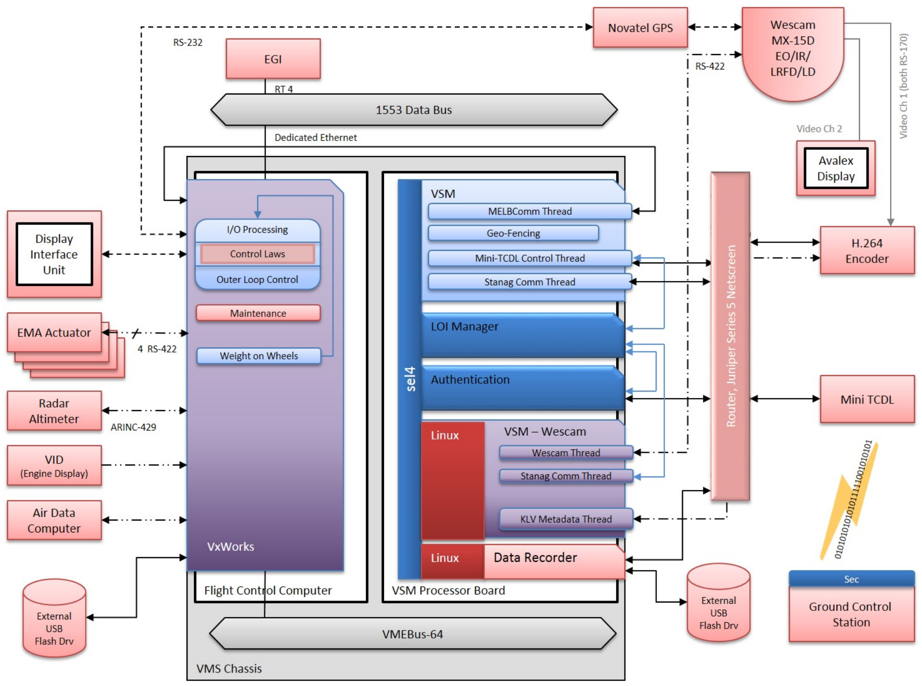


Figure 52: ULB HACMS Final Architecture

mented using HACMS technologies, more detailed software architecture models were developed.

In the case of the VSM, four major changes were made to the baseline VSM architecture. First, the original single VSM was split into a "flight" VSM and a "camera" VSM. The flight VSM managed control of the aircraft, including waypoint processing and communication with the FCC. The camera VSM controlled the Wescam sensor. Ground station commands associated with waypoint navigation and aircraft state would be routed to the flight VSM. The flight VSM is therefore regarded as more critical and it was completely reimplemented using HACMS technology. The camera VSM is less critical, and both for pedagogical reasons and due to resource constraints, was retained largely as legacy code running in a Linux partition with seL4 serving as hypervisor. The data recorder function was likewise retained as legacy code in a separate Linux partition. These Linux partitions could then serve as "jumping off points" for Red Team attacks.

The second change, in part a result of the baseline Red Team assessment, was the introduction of an authentication component to authenticate communication with the ground station, and ensure only commands from an authenticated ground station would be acted upon. The same Galois Counter Mode authentication algorithm was used on both the ULB and the research air vehicle. For the ULB ground station, an authentication shim was added between the ground station and the ground side radio (to avoid modifying the ground station software). The third change was the introduction of an LOI Manager component between the authentication component and the two VSMs. The LOI manager enforces the STANAG protocol rules on which commands can be performed by a ground station with a particular LOI level.

The fourth change to the VSM was the addition of a geo-fence capability. With the proliferation of COTS UAVS, geo-fencing has been seen as a way of preventing (inadvertent) "rogue drones." A geo-fence defines a particular region as being prohibited for the UAV. In addition to these civil applications, geo-fences also have military application where they represent operational area boundaries, no fly zones, threat regions, and pop-up threats. For the ULB, the geo-fence capability is limited to a simple two dimensional rectangle, and is enforced both at the route level, and dynamically. At the route level, when a waypoint route is uploaded to the VSM, it is checked against the geo-fence, and a route which would violate the geo-fence is rejected. Dynamically, as the aircraft is flying, its current position and velocity is checked against the geo-fence, and if a violation is imminent, the ULB is instead automatically directed to loiter within the geo-fence. This dynamic check is necessary because of the presence of autopilot and slave-to-sensor navigation modes. Figure 53 shows dynamic geo-fence enforcement.

The FCC architecture was essentially unchanged. In both the baseline and final HACMS ULB FCC, there was essentially a single component with three threads, the real-time control thread, a background thread, and a logging thread.

Detailed software AADL models were developed for both the VSM and the FCC. An AADL model defines a system in terms of the hardware and software

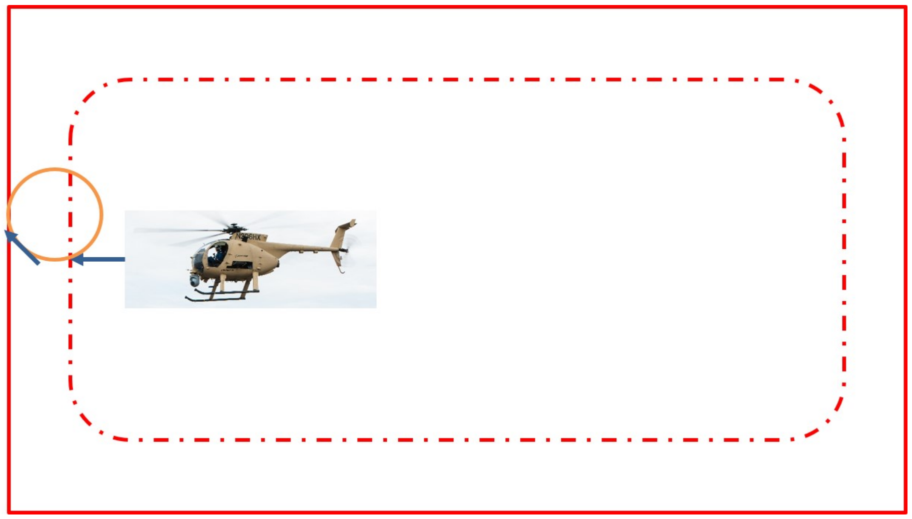


Figure 53: Dynamic Geo-Fence Enforcement

components, and the connections between them. An architecture model has an architectural style which, generally speaking, defines the types of interfaces provided by components, and the semantics of the connections between them. In the case of the HACMS ULB, there were two distinct architectural styles, one for the FCC and one for the VSM.

The FCC used an AADL architectural style compatible with VxWorks, in particular there was really only a single component hosting the three threads, so the semantics of component communication were unaddressed. The AADL architectural style for the VSM is influenced by, but is not identical to, the component semantics provided by CAmkES, the component model associated with seL4. Threads could be active or passive, and communication is via shared data or by event data ports, corresponding to seL4 communication primitives. There were two special cases for the VSM model. First, the VSM model included provisions for components representing virtual machines running in a partition, with specialized connections across the virtual machine boundary. Second, some seL4 platform functionality, such as the UDP stack, was provided by assemblies of CAmkES components. These are CAmkES components not AADL components, and so have slightly different semantics. These are represented in the HACMS VSM AADL architectural style through the introduction of *external components*. These are AADL components that essentially encapsulate the native CAmkES assembly and bridge the gap between architectural styles.

Once the AADL model is defined, it can be used for verification and for code generation. Verification is addressed in section 4.1.1. Code generation was implemented in the University of Minnesota AADL2RTOS tool. The architectural style is specified by setting the system OS type in the AADL model, and the tool generates the appropriate glue code. In the case of VxWorks, the glue code is the appropriate VxWorks configuration information. In the case of seL4 and CAmkES, the AADL2RTOS tool generates CAmkES ADL and IDL files, and then CAmkES generates component skeletons from the ADL and IDL files. The AADL2RTOS tool also generates code that connects the CAmkES skeletons with the component implementations that are specified in the AADL model.

9.2.2 Components

Two HACMS component implementation technologies were used on the ULB. Most of the HACMS components were implemented using Ivory, the EDSL developed by Galois (Section 5.1). Using Ivory guaranteed that those components were free from a variety of implementation defects, in particular, various memory issues (null pointers, buffer overflows, array index errors), since the design of the language does not allow developers to write code with these defects. The second component technology used was Spiral/Hybrid Control Operator Language (HCOL), which uses formally verified transformations to generate correct by construction component implementations from high level algorithm descriptions. Spiral/HCOL were developed by Carnegie Mellon University and SpiralGen. Spiral/HCOL were used to implement an air/ground state determination algorithm (”weight on wheels”) during HACMS Phase 3, as shown in Figure 52.

Ivory Ivory was designed for writing cyber-physical system software. As an EDSL embedded in the Haskell functional language, the standard Ivory syntax is Haskell-like. Ivory also provides a ”concrete syntax” that is more C-like. Code written in Ivory is compiled into C, which is then compiled using an off the shelf C compiler. The Ivory compiler, written in Haskell, enforces the formal properties of the Ivory language, so that though the executing code is C, the generated C is guaranteed to be free from the defects Ivory is designed to prevent.

Ivory builds code as modules. Each module specifies the dependencies on other modules or external resources, and is the bridge between Ivory and Haskell, including all the various Ivory language EDSL definitions. The Haskell compiler is then used to compile the module, which produces an executable that includes the Ivory compiler as well as the code for the module. Executing the module then executes the Ivory compiler, which generates C code implementing the application code of the module. This generated C code is then compiled for the target system and included as part of the final executable system.

For the ULB, the generated C code provides the implementations of the software components in the ULB AADL model, since the AADL model is the basis for the system design. For the AADL Ivory components, the generated C code is associated with the appropriate AADL components using AADL properties. The properties are used by the University of Minnesota’s AADL2RTOS

```
void LOIComponent::AckAdd(int idx, unsigned int msg)

    for (int jj = 0; jj < 4; jj++)

     if (MCucsAcks[idx][msg%37][jj] == 0)

     MCucsAcks[idx][msg%37][jj] = msg;
     return;
```

Figure 54: Original C++ Code

tool to invoke the component implementations from the generated component skeletons.

Since the ULB is a legacy system, the approach taken to develop the Ivory implementation of ULB components was to essentially adapt the existing software to Ivory rather than to try and develop a new VSM from first principles. For the VSM, the first step in the adaptation was the refactoring described above into separate flight and camera VSMs, and separate authentication and LOI manager components. The Authentication and LOI components were implemented by Boeing in collaboration with Galois.

The remaining VSM Ivory components, and the FCC Ivory components, were implemented by Boeing. The second step in implementing the VSM was to modify the remaining legacy code to make it more Ivory-like. The existing ULB implementation is an object oriented C++ application that makes use of a generic avionics component framework, while the Ivory programming model is more like that of statically allocated imperative C. These modifications included de-objectifying the existing software, for example by replacing polymorphism with explicit message types in the message handling library. They also included transitioning from dynamic memory allocation and dynamic data structures to static sized data structures. The FCC software was already essentially in this form. At that point, the various components could more or less naturally be rewritten using the Ivory concrete (C-like) syntax.

Figures 54-56 provide a comparison between C++ and Ivory implementations of a simple function, and the C code generated from Ivory.

The ULB FCC consists of three tasks: foreground, background, and logging. The foreground task itself contains three functions: I/O processing, outer loop control, and inner loop control. Two of these functions were implemented using Ivory: the outer loop waypoint control, and the input/output interface code. These elements of the code are both separable from the inner control loop, and represent the threat surface of the FCC software (since the inner loop can only be reached through the outer loop or the device interfaces), and so represent an ideal FCC subset for the application of SMACCM technology. The inner loop

```
void loiAckAdd(CucsIdx idx, uint32_t msg)

        let ackhash = (&MCucsAcks)@idx;
        let bucket = ackhash@toIx(twosCompCast(msg));
        map jj
                if (*bucket@jj == 0)
                        store bucket@jj as msg;
                        return;
                  else
```

Figure 55: Ivory Concrete Syntax

```
void loiAckAdd(int32_t n_var0, uint32_t n_var1)

    uint32_t(* n_let0)[4U] = MCucsAcks[n_var0];
    uint32_t* n_let1 = n_let0[(int32_t)
        ((bool) ((uint32_t) (n_var1 >> (uint32_t) 31U) ==  (uint32_t) 1U) ?
            (int32_t) ((int32_t) -(int32_t) (uint32_t) ~n_var1 - (int32_t) 1)
            : (int32_t) n_var1) % 37];

    for (int32_t n_ix2 = (int32_t) 0; n_ix2 <= (int32_t) 3; n_ix2++)
        uint32_t n_deref3 = n_let1[n_ix2];

        if ((bool) (n_deref3 == 0))
            *&n_let1[n_ix2] = n_var1;
            return;
```

Figure 56: Generated C Code

control was retained as legacy C code that was invoked by the Ivory code.

Implementing the outer loop control in Ivory was comparable to implementing the Ivory VSM components. The I/O processing code however presented some unique aspects. The legacy ULB I/O code uses non-volatile random access memory (NVRAM). Legacy NVRAM code is written so that all access to NVRAM happens as 32 bit words on 32 bit boundaries, but there was still a need to store non-word data such as strings. Using only the concrete syntax, we would have had to convert the default static string into 32 bit words at runtime before writing to NVRAM. Using the Ivory Haskell syntax, Haskell can do such conversions at compile time and only generate code that writes the converted data. The other issue with the NVRAM data is that it is memory mapped to a specific location in memory. Ivory doesn't provide a way to set a pointer to globally allocated data to a specific address. In this case, we used the Ivory extern interface to define the pointer location in a static C file and imported that definition into the Ivory code.

Another issue arose in interacting with the aircraft hardware. We were able to use the Ivory serialization library to do the necessary byte operations in a type checked manner without using extraneous runtime casting operations. However, VxWorks hardware interfaces required use of more complex interfaces with external libraries than the simple message passing and remote procedure calls (RPC) of seL4. These C interfaces require more extensive importing of functions into the Ivory domain. This process is a source of potential error. For example, most of the C interfaces use a NULL pointer to represent arguments without a valid value. Ivory intentionally does not allow NULL pointer values (to prevent memory errors) so this required a work-around. Instead of using NULL, we specified the arguments as unsigned 32-bit values. Passing 0 provides an effective NULL to the underlying C code. The same approach was used for returning pointer values to the OS (i.e. stdio file operations).

SpiralGen HCOL and Spiral were developed by Carnegie Mellon and SpiralGen for generating high assurance control code. HCOL is a high level language used to specify control algorithms, and uses mathematical concepts used by control engineers. Spiral is a multi-stage re-writing system for performing correct code synthesis from HCOL algorithms ([59]). We used HCOL and Spiral to implement an air/ground state estimation algorithm (more commonly known as weight on wheels, or in the case of the ULB, weight on skids). This is a safety critical function, especially for an unmanned vehicle. If the aircraft is in the air and the software believes it is on the ground, it may result in loss of aircraft. If the ULB is on the ground and the software believes it is in the air, dynamic rollover can occur, which can also result in loss of the vehicle.

In collaboration with Carnegie Mellon and SpiralGen, we used HCOL and Spiral to develop a high assurance weight on skids component. We began with a simplified algorithm and proceeded to a more sophisticated one. The resulting software was integrated as a library within the FCC. Due to safety of flight concerns (since demonstrating a compromised algorithm near the ground was judged to pose unnecessary risk even with the safety pilot on board to override the VMS), the weight on skids software was not included in the HACMS FCC software flown on the aircraft. Instead, the high assurance weight on skids component was demonstrated in simulation.

We have demonstrated the impact of a compromised weight on skids algorithm using the Boeing ULB simulator. If an adversary is able to cause the weight on skids algorithm to produce the wrong result, simulated aircraft can lose control when it is airborne, but mistakenly believes it's on the ground. In the scenario we tested, this occurs because the stability/control gains on the ground are lower than they are in the air. This is a conventional design approach; lower gains on the ground are required to ensure stability on the ground, and higher gains in the air are required to maintain stability during maneuvering. When the helicopter is airborne, but incorrectly thinks it's on the ground, the lower gains are unable to maintain the stability of the aircraft in flight. Note that we posited a successful exploitation of a vulnerability in the algorithm, and did not attempt to create an exploit that would produce the incorrect output. We were able to show that the HCOL/Spiral version of the weight on wheels algorithm correctly handled a simulated attack that induced a floating point error.

9.2.3 Kernel

The formally verified seL4 microkernel was used on the ULB VSM as the foundation for the security enhancements provided by HACMS. seL4 provides a number of security benefits. First, since the correctness of the kernel has been formally verified, the OS kernel is no longer part of the attack surface of the aircraft, at least with respect to software defects (so for example attacks that exploit hardware defects or similar vulnerabilities, such as Rowhammer or a cold boot attack, are not included). Second, seL4 provides guaranteed memory and time isolation between memory spaces. The memory isolation requires hardware support. On x86 architectures, this includes VT-x instructions and EPT virtualization extensions. The original ULB VSM hardware (a relatively old Pentium-M) did not provide the required hardware support, so was replaced for HACMS with a Core i7 processor.

For the most part, ULB use of seL4 was indirectly through AADL and CAmkES. CAmkES, the Component Model for microkernel-based Embedded Systems, is the seL4 component model. CAmkES includes both an Interface Definition Language for specifying the interfaces of CAmkES components, and an Architecture Definition Language for specifying how the components are connected. CAmkES is comparable to the CORBA component model. This component model is what is targeted by AADL2RTOS.

AADL2RTOS does so by adapting the HACMS AADL component computation model to the seL4 CAmkES computation model. As an example, AADL2RTOS generates glue code that adapts the AADL threading model to the CAmkES threading model by implementing monitors for the AADL components, and generates seL4 synchronization primitives to implement AADL synchronization mechanisms. As discussed earlier, AADL external components are used to represent native CAmkES assemblies, such as those used to implement services such as UDP.

In addition to building systems using CAmkES components, seL4 can also serve as a hypervisor hosting guest operating systems For the ULB, the guest operating system was Linux. Communication across the virtual machine boundary was provided using an adaptation of the vchan library developed for communication between different VMs on the same Xen host. For the ULB, there is one channel between the AADL components and the camera VSM VM, between the AADL components and the data logger VM, and one channel between the camera VSM VM and the data logger VM. In the case of the AADL component side, this will take the form of an external vchan adapter component. The AADL components that communicate with either the data logger or the camera VSM will exchange messages (remote procedure calls or events) with the adapter component. Elements inside the VMs, either the data logger or the camera VSM, communicate using seL4 supplied libraries providing the Linux side of the vchan protocol interface.

9.2.4 Build Process

The build process (Figure 57) for the ULB VSM for seL4 involves five tool families. Ivory tools are used to transform Ivory source code for component implementation into C source code. The AADL2RTOS tool transforms the AADL model into glue code and CAmkES artifacts. The CAmkES tool chain is used to generate seL4 component implementations and glue code from CAmkES ADL and IDL files. Other tools are used to build VM images from the source files for the camera VSM and the data logger. Finally, the seL4 build tools are used to combine the various application source files, the VM images, and seL4 source files into the final executable seL4 image for loading onto the VSM computing hardware. The build process for the FCC is comparable (with the absence of the VMs).

There are three sets of user created source files for the VSM application. The first are the source files for the VM applications. These are the modified legacy source files for the camera VSM and the data logger. The second set of source files are those that make up the AADL model of the system. The AADL model includes 6 files for the detailed VSM software model, and an additional AADL file for the simpler FCC software model. There are two types of Ivory files, concrete syntax files and Haskell syntax files. The Ivory tool chain compiles those source files into an executable that then is used to generated C source files. The generated C source files interface with the AADL2RTOS generated glue code.

9.3 Demonstrations

Two flight demonstrations of the ULB were carried out during the HACMS program. A risk reduction flight of the Phase 2 configuration (Figure 51) on

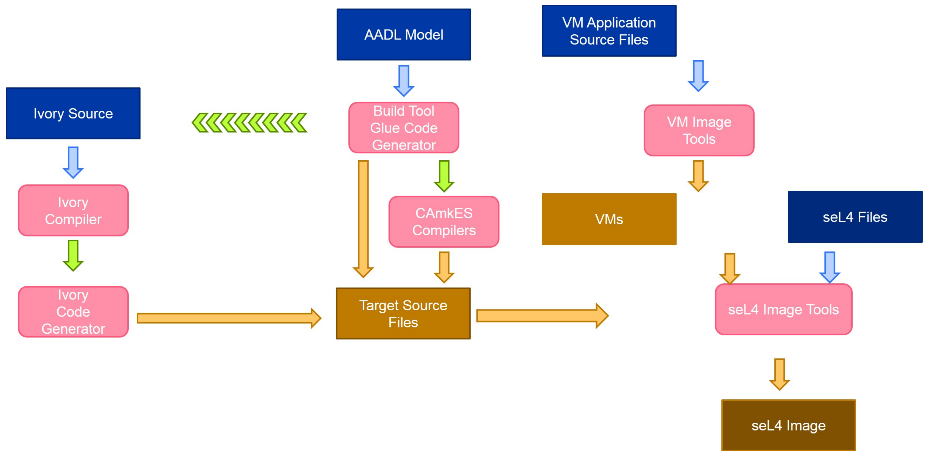


Figure 57: ULB VSM Build Process

24 July 2015, and the final demonstration of the Phase 3 configuration (Figure 52) on 9 February 2017. Both flights originated at the Boeing Mesa facility with the ULB flying as an optionally manned aircraft with onboard safety pilot under an FAA experimental aircraft license. All testing was conducted using the Unmanned Little Bird aircraft, N206HX.

9.3.1 Initial Demonstration

The HACMS ULB demonstration flights required approval from AFRL. In order to reduce risks associated with both the HACMS technologies and the flight demonstration approval process, we performed a risk reduction demonstration flight of the ULB at the end of Phase 2. The demonstration involved two ULB flights, a check flight on 21 July 2015, and the live demonstration flight on 24 July. These flights demonstrated that the ULB was still functional using the Phase 2 Architecture, where the original VSM software was replaced by components implemented in Ivory running on seL4, integrated via glue code generated using AADL2RTOS. These flights were conducted from the Boeing Mesa facility, the ULB's home field.

The risk reduction demonstration flight was coordinated with Boeing Test and Evaluation and AFRL. We began coordination with AFRL in November of 2014, working with them to identify the appropriate process and the information necessary to receive approval for the flight demonstration. We provided information about the usual ULB flight procedures and sample documentation from previous ULB demonstrations, and reviewed them with the AFRL team.

A Boeing flight demonstration package typically includes an Engineering Test Request, identifying the requirements and objectives for the test, a Safety Plan, a Test Hazard Analysis, Test Plan, Test Cards, and a Safety of Flight Review. The Boeing documentation included substantially all the information that was required for AFRL approval. With that confirmation, we created the Boeing documentation required for the July 25 flight demonstration.

The AFRL process for flight test is defined by AFRL MANUAL 99-103, AFRL Flight Test and Evaluation. The AFRL team led compliance with the approval process, and Boeing provided the information as it was generated or requested. This included completing AFRL form AFI 91-202_AFRLSUP1, the Flight Activity Information Worksheet, which contains the high level information about a flight demonstration. AFRL used the information provided by Boeing to prepare for and brief an AFRL Technical Review Board (TRB) on 26 June, which approved the demonstration, once some additional information was provided, including planned flight times and data to be collected. Since the ULB is a Boeing owned asset, the demonstration flight was not being conducted on a USAF range, and was being conducted under the existing ULB FAA license, a determination was made that no formal AFRL Safety Review Board (SRB) was required. Final approval for the flight demonstration was received from AFRL on 20 July.

The first demonstration flight was performed on 21 July. The formal demonstration flight occurred on the morning of 24 July, 2015 (Figure 58). This flight demonstrated that the ULB with HACMS VSM software still flew as expected, and showed that increasing the cyber security of military vehicle systems could be done without compromising (soft) real-time performance.

9.3.2 Final Demonstration

The objective of this demonstration was to verify that the FCC and VSM software re-architected using HACMS technology is functionally the same as the pre-HACMS software, yet is more secure, and provably more secure. The demonstration was planned to include two sorties. The first sortie showed the non-HACMS ULB software vulnerable to a targeted software attack. In this scenario a virus was intentionally introduced which would assume control of the aircraft's EO/IR sensor. For the second sortie the aircraft flew configured with HACMS software. The first half of the second sortie would demonstrate that HACMS software component partitioning has rendered the virus ineffective. The second half of the sortie would then show how the HACMS geo-fencing component can be used to keep the aircraft from violating airspace restrictions when a simulated *supply chain* attack originates from the EO/IR sensor itself. The continued real-time performance of the HACMS configured aircraft, and its success in resisting the two attacks, shows that cyber security does not have to come at the expense of CPS performance.

The first attack was representative of a compromised maintenance device as an attack vector. A compromised maintenance device was connected to the USB socket on the VSM that normally hosts the USB drive used by the data logger. The device then injected a virus which pivots from the data logger to the VSM, and causes the Wescam sensor to be stowed. This attack is successful on the baseline aircraft which has no internal protections between components. The attack was unsuccessful on the HACMS configured ULB since seL4 memory

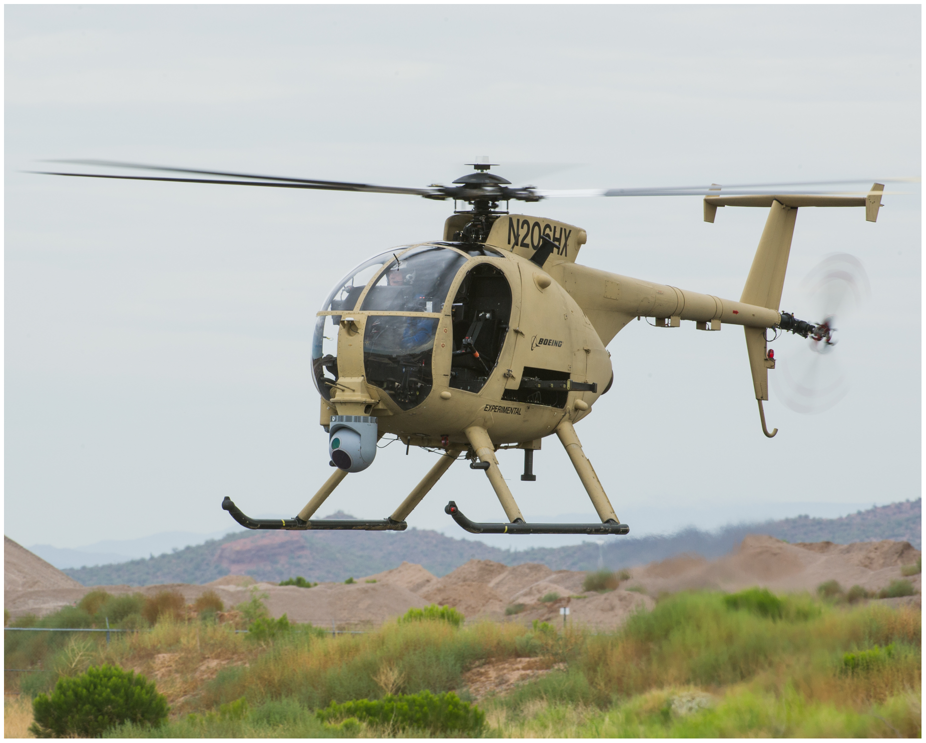


Figure 58: ULB Risk Reduction Flight Demonstration 24 July 2015

Figure 59: ULB Final Demonstration 9 February 2017

protections confine the virus to the data logger partition, where it could not affect the camera (or other aircraft flight functions).

The compromised maintenance device and virus were created by the HACMS Red Team. The compromised maintenance device was a Raspberry Pi Zero. The virus was a variant of the DuckberryPi with a payload configured for the ULB. DuckberryPi is a Raspberry Pi distribution which causes a Raspberry Pi to act like a USB based keyboard emulator and automator. When inserted into the USB port of a computer, the Duckberry Pi masquerades as a USB keyboard and sends the keystrokes in its payload as keyboard inputs to the computer. The payload for the ULB malware repeatedly sends a Stow command to the Wescam, resulting in mission failure.

The simulated supply chain attack on the aircraft (foiled by the geo-fence) was implemented by Boeing and involved modifying the Boeing Wescam VSM code to use fixed coordinates outside of the geo-fence for the stare point whenever the slave to sensor mode was engaged. This emulates compromise of the COTS camera software, which could have been achieved by supply chain compromise.

There are two strands to the geo-fence attack. First the GCS operator will attempt to upload a route that violates the pre-programmed geo-fence. The VSM will reject the route. The aircraft will then be placed into slave to sensor mode. In this mode, the aircraft loiters about the stare point of the EO/IR sensor. When the aircraft enters slave to sensor mode, the VSM software will begin generating a fictitious sensor stare point to emulate a supply chain based attack on the aircraft. The malicious stare point will be centered on Sawick Mountain just outside of the geo-fence boundary. The continuous real-time checking of the aircraft state by the geo-fencing component will command the aircraft to enter a stationary loiter before the aircraft violates the geo-fence. At this point the GCS operator will command the aircraft to return to base and the demonstration will be complete.

The approval process for the final demonstration echoed that of the Phase 2 demonstration. An AFRL TRB was held on 14 November 2016, which approved the demonstration, once some additional information was provided, including planned flight times and data to be collected. Since the ULB is a Boeing owned asset, the demonstration flight was not being conducted on a USAF range, and was being conducted under the existing ULB FAA license (renewed 14 December 2016), a determination was made that an informal AFRL Safety Review Board was sufficient. The SRB was held on 20 January 2017. Final approval for the flight demonstration was received from AFRL on 24 January 2017. Verification, practice, and demonstration flights were conducted 25 January through 9 February (Figure 59).

10 Results and Discussion

This section presents some lessons learned over the course of the HACMS program, as well as several recommendations for future work.

10.1 Type-Checking for Embedded Programming

Build times are non-trivial for large software systems. At the time of writing, a fresh build of SMACCMpilot and associated test programs is over seven minutes of real time (and 12 minutes of CPU time since we have a multi-threaded build system). One reason the build time is so large is that it requires Cabal (the Haskell package manager) to discover library dependencies and install packages, compile the Haskell sources, and then compile the C sources. As well, some sources are compiled multiple times for different targets on multiple operating systems.

Then, to execute the software on the embedded device, we have to write the software to the device's memory via a Joint Test Action Group (JTAG) programmer or a serial boot loader, which takes on the order of ten seconds.

All this is to say that the end-to-end debug cycle might mean testing a small number of changes to Ivory or Tower per hour. Clearly, the debug cycle in embedded development particularly motivates us to make fewer bugs and to discover them early.

During development, it became apparent how useful Haskell type-checking is for embedded programming. As described in Section 5.1, we have embedded Ivory's type system in Haskell's. Thus, domain-specific type-errors are caught during Haskell type-checking. Type-checking, and other static warnings reported by GHC, are nearly instantaneous since it can be done on a module-by-module basis. The type system tracks the global or stack frame provenance of references, as well as structure accessors and array indices, to ensure all well-typed Ivory programs generate memory-safe C. The upshot is that Ivory programs that would generate unsafe C programs are caught immediately.

In addition, we have found it useful to detect potential bugs even if the C compiler might also detect them. To take one example, consider unused variable declarations. While a C compiler can detect this, perhaps late in the compilation phase, we discover these warnings nearly instantaneously during type checking. Moreover, the more preprocessing we can do in Haskell, the more potential errors we may find, and with a better relation to the EDSL source.

However, not every property of interest in embedded programming can be conveniently embedded in the Haskell type system with GHC extensions. For example, integer overflows checks are not practical to embed.

Moreover, GHC's type error reporting can be unwieldy. Ivory users would benefit from domain-specific error reporting which could, for example, describe type errors in the vocabulary of Ivory, rather than burden the user to interpret the way the Ivory language types are embedded into Haskell types. For example, passing the wrong number of arguments to an Ivory function in a procedure call is reported as a type error when using functional dependencies, whereas a mismatch between the type of a procedure and the number of arguments provided in its declaration is reported as a kind error. The errors reported are of the particular type-level implementation given for Ivory types. Haskell does not yet have good facilities for type-level programming abstraction.

10.2 Type-Safe System Plumbing

Adding many new features to SMACCMpilot is easy. In fact, the most tedious part is writing the business logic in Tower, where we define a new task, and then plumb values representing communication channels through the code. There is nothing conceptually difficult in doing so—it is similar to any monadic interface for specifying a graph. When changes cross Haskell function boundaries, we must modify the arguments to the Haskell function that generates the Tower task (or modifying the fields of a data-type if channels have been grouped together). Channels are typed, so type-checking detects most plausible inter-task communication errors.

Stepping back, the idea that plumbing arguments to Haskell functions is the hardest part of embedded development is amazing. We are not dealing with bugs in low-level OS interfaces, we are not making timing or resource contention errors in communication, we are not dealing with type-errors like you might find in raw C (where data might be cast to `void*` or `char[]`).

Because plumbing is so easy, it encourages us to improve modularity in the system. Defining a new RTOS task is easy, so we might as well modularize functionality to improve isolation and security. For example, in the ground station communication subsystem, encryption and decryption are each executed in isolated tasks, simplifying the architectural analysis of the system.

10.3 Faking a Module System

In Ivory and Tower, top-level functions and structures are packaged into a Haskell data structure to provide to the Ivory compiler. The onus is on the programmer to package up all the necessary components.

On one hand, the approach provides the programmer control over how to modularize the generated C code, deciding which definitions to put in a C source or header file. On the other hand, we have found it to be verbose, tedious, and error-prone. Generally, we want the C files to have similar structure to the Haskell modules in which Ivory programs are written. From that respect, the Ivory module system simply duplicates the Haskell module system.

Worse is when the programmer forgets to package a definition. The error only becomes apparent at C *link* time, near the end of a long build process. Missing definitions have plagued our builds.

We could move symbol resolution up the build cycle to the C-code generation phase. Ideally, we would move it up the build cycle even further. We are currently exploring the use of Template Haskell to generate Ivory modules at compile-time to assist the programmer.

10.4 Control Your Compiler

If we were writing our application in a typical compiled language, even a high-level one, and found a compiler bug, we would perhaps file a bug report with the developers... and wait. If we had access to the sources, we might try making a change, but doing so risks introducing new bugs or at the least, forking the compiler. Most likely, the compiler would not change, and we would either make some *ad-hoc* work-around or introduce regression tests to make sure that the specific bug found is not hit again. Such a situation is notorious in embedded cross-compilers that usually have a small support team and are themselves many revisions behind the main compiler tool-chain.

But with an EDSL the situation is different. With a small code-base implementing the compiler, it is easy to write new passes or inspect passes for errors. Rebuilding the compiler takes seconds.

More generally, we have a different mindset programming in an EDSL: if a class of bug occurs a few times—whether caused in the compiler or not—we change the language/compiler to eliminate it (or at least to automatically insert assertions to check for it). Instead of a growing test-suite, we have a growing set of checks in the compiler, to help eliminate both our bugs and the bugs in *all* future Ivory programs.

We claim that Ivory code compiles to memory-safe C code. However, a formal proof of these claims, or more generally, a proof that the semantics of Ivory programs are implemented by the generated C code, is work-in-progress. However, a small number of primitives and simple compiler facilitates inspection and testing. In all, this is less assurance than is given by fully verified toolchains, such as CompCert [60]. Other approaches more specific to bringing assurance to EDSL compilers could be borrowed as well [61].

10.5 Everything is a Library

With an EDSL, and particularly a Turing-complete macro language, everything is a library. The distinction between language developers and users becomes ambiguous. As an extreme example, one can think of Tower as "just" a library for Ivory. A small example is defining a conditional operator in terms of Ivory's if-then-else primitive as shown in Figure 60. All types above were introduced in Section 5.1. With the `cond_` operator, we can replace nested if-then-else statements as shown in the figure with more convenient conditionals, without modifying the language.

Because macros are so easy to define and natural in EDSL development, our biggest challenge has been ensuring developers on our team put useful ones in a standard library, to be shared.

10.6 Semantics

To take advantage of legacy cross-compilers, we are forced to generate C code from our EDSL. A large focus in designing Ivory is to allow expressive but

```
data Cond eff a =
  Cond IBool (Ivory eff a)

(==>) :: IBool -> Ivory eff a
      -> Cond eff a
(==>) = Cond

cond_ :: [Cond eff ()]
      -> Ivory eff ()
cond_ [] = return ()
cond_ ((Cond b f):cs) =
  ifte_ b f (cond_ cs)
```

```
cond_
  [ x >? 100 ==> ret 10
  , x >? 50  ==> ret 5
  , true     ==> ret 0 ]

ifte_ (x >? 100)
 (ret 10)
 (ifte_ (x >? 50)
   (ret 5)
     (ret 0))
```

Figure 60: Conditional Ivory macro.

well-defined programs. We believe Ivory cannot produce memory-unsafe C programs. However, undefined C programs can be generated from Ivory; for example, signed integer overflow and division-by-zero are undefined. Guaranteeing programs are free from these behaviors is decidable (the arithmetic is on fixed-width integer types), but intractable to prove automatically.

To assist the programmer, the Ivory compiler automatically inserts predicates into the generated code to check for overflow, division-by-zero, etc. The user defines the behavior of the program if a check fails. For example, during testing, we define the checks to insert a breakpoint for use with a debugger. Another option may be to do nothing and rely on the semantics provided by the C compiler. Still another option might be to trap to a user-defined exception-handler. Currently, SMACCMpilot contains approximately 2500 compiler-inserted non-trivial checks that cannot be constant-folded away. In the future, we hope to *prove* these checks never fail.

Early in the development process, we used the C Bounded Model Checker (CBMC) model checker [62] to partially verify the assertions in the generated C code. However, as our application grew, we ran into three problems. First, a naive application of whole program model checking did not scale to our application size. Second, many assertions depend on user-provided preconditions (e.g. on inputs from hardware devices). Third, some assertions were undecidable (e.g. non-linear arithmetic).

There are two other semantics categories to consider: defined behavior and implementation-defined behavior. In Ivory, we attempt to eliminate almost all implementation-defined behaviors. For example, only fixed-width size types, like `uint8_t` or `int32_t`, can be generated. Implementation-defined sizes, like `int` or `char` are not used. We have found these to be dangerous: programmers might assume properties about the size of a type that do not hold in a non-standard architecture (e.g., that an `int` is at least 32 bits or that `char` is unsigned;

both are implementation-defined). Such assumptions are particularly dangerous when porting code between different embedded platforms. Indeed, when we ported portions of ArduPilot, initially built for an 8-bit AVR architecture to a 32-bit ARM, we found these sorts of implicit assumptions.

Finally, even defined behavior is not necessarily intuitive behavior. For example, in C, the defined behavior for arithmetic on values that have a size-type smaller than `int` is to implicitly promote them to `int`s before performing the arithmetic.

For example, given

```
uint8\_t a = 10;
uint8\_t b = 250;
bool     x = a-b > 0;
bool     y = (uint8\_t)(a-b) > 0;
```

`x` evaluates to 0 and `y` to 1, provided that

```
sizeof(uint8\_t) < sizeof(int)
```

This behavior is worrisome to the embedded programmer because, across various embedded processors and C compilers, integer sizes are often defined differently.

In Ivory, arithmetic is at the size of the operands, which we believe is more intuitive. We force the generated C to respect this semantics by inserting casts into expressions. So the Ivory expression `a-b` results in the C expression `(uint8_t) (a-b)`.

10.7 Integration Problems

There were several issues outside of the architecture description model that caused problems during build and integration, cumulatively requiring both several calendar months and person months to fix. These issues suggest that although the guarantees provided by a trusted OS, build integration support through an architecture description language, and a type-safe programming language, there are still integration issues that are problematic and need to be addressed to create a high-assurance system build. These issues involve integration between multiple build systems, legacy and auto-generated code, support for multiple platforms, and issues when using C in a high-integrity build process.

10.7.1 Drivers

The area that was the single most significant in terms of debugging time, schedule slippage, and risk involved the driver software on the various deployment platforms. Although the seL4 kernel is guaranteed to be correct by virtue of its proof, similar guarantees cannot be provided of drivers, because they depend on hardware vendors to provide accurate formal models of the behavior of the device, which usually does not occur. In fact, even the informal spec sheets often contain errors for the hobbyist microcontrollers that we were using to create the software.

The most significant issue involving drivers occurred in the second phase of the project, when a high-priority communication driver thread did not properly clear its interrupt, which in certain cases caused repeated raising of the same interrupt leading to schedule slippage and eventual failure of the SMACCM-copter. This issue only manifest when a certain level of load was placed upon the system, so did not show up until we loaded the full Phase 2 software into the hardware. This issue was not diagnosed and fixed until two weeks before the Phase 2 demo, and required at least 1 person month to diagnose and fix.

There were several other driver issues involving, for example, misconfigured timing: in one case the seL4 driver ran 33% slower than clock time, and in another case it ran just under 10% slower than clock time, leading to incorrect performance of several algorithms within the system. There were also a handful of issues related to deadlocks and race conditions in drivers.

10.7.2 Legacy/Untrustworthy Code Support

The open-source platform involves both C code autogenerated from Ivory and legacy code. This is by necessity; it was not reasonable for Galois to rewrite the entire Arducopter platform into Ivory in 18 months, though a considerable amount was converted. This legacy code caused several problems for our architecture-driven approach; it is arguably a root cause of many of the problems we experienced. First, the legacy code was not well structured and did not conform to the communication approaches supported in AADL, so it was not straightforward to introspect the structure of it. To support these in AADL, we added support for external threads in AADL, which did not follow the standard AADL dispatch structure, as well as named external semaphores and mutexes. However, the communication structure involving these threads was not represented in the AADL model, leading to fidelity issues between the implementation and architectural model.

In addition, the legacy code was originally designed for the FreeRTOS operating system and expects the API for this platform. The NICTA team built a "shim" to mimic the FreeRTOS OS API for the legacy code, but the underlying principles of the OSes are quite different. The most notable difference is that eChronos is statically configured: all tasks, mutexes, and semaphores have to be pre-declared in a configuration file; FreeRTOS is dynamically configured; these constructs are created after the program starts. To support the FreeRTOS API, NICTA created a scheme where mutex and semaphore ids following a specific naming convention would be parceled out by the FreeRTOS shim. Unfortunately, the initial version did not work correctly in some cases, so situations could occur in which multiple logical semaphores were bound to the same OS semaphore, causing the system to occasionally deadlock, when the tasks using the logical semaphores would interact.

In subsequent phases, we worked much harder to modularize dependencies to legacy code. First, Galois was able to replace the blob in Phase 2 with an entire Ivory-based flight controller. However, there was still substantial interaction with untrusted code involving the camera and tracking software. For this, we used a virtual machine with mediated access to the rest of the system, which worked very well. In fact, as described in the SMACCMpilot description, we were able to allow a malicious attacker to start a forkbomb inside the VM without disrupting flight operations.

10.7.3 Make Issues

The SMACCM project inherited a complex make system that had been built to work with a FreeRTOS version of the SMACCMcopter. Our original plan was to generate the final makefile from the AADL model, but this turned out to be impractical. The most significant issue had to do with the dependencies in the make process. The basic outline is shown in Figure 34: the Ivory/Tower code generates a portion of the AADL model, as well as C code. The AADL model generated the eChronos OS using a configuration file, and both the Ivory-generated and legacy C code are dependent on the generated eChronos headers. The legacy code has a complex make system of its own. Our plan was to start final build from AADL, importing the legacy code as a library, but because eChronos was built from the AADL model and the legacy code was dependent on the eChronos headers, it was not straightforward to do this.

We experimented with adding various make-related properties to the AADL file in order to perform the legacy build steps from within AADL but ultimately discarded this plan; it was complex and seemed to add little value to the AADL model. Instead, we modified the manually created makefile to include the AADL build step to construct the eChronos OS. However, because the makefile is not derived from the architectural description, there are many C files that are pulled into the build that are not visible in the model. A principled solution remains the subject of future work.

10.7.4 Exceeding Static Memory when Generating System Image

After an update of the Ivory components to add additional flight modes, our binary started immediately crashing upon start on the target platform. After debugging, we realized that we were exceeding the available SRAM. The ARM Cortex-M4-based ArduCopter has 192KB of SRAM that is divided into three segments of 112KB (S1), 16KB (S2), and 64KB (S3). The gcc/gdb toolchain does not signal an error if the available memory is exceeded, and our expectation was that we were not close to the memory ceiling. In addition, only segments S1 and S2 are continuous, so we effectively had 128KB, rather than 192KB in which to fit the data for the system binary. After examining the binary, we realized that the makefile was including both statically allocated memory for communication primitives used by the Tower-FreeRTOS build as well as memory for the AADL-generated communication primitives, effectively doubling the required SRAM for communications. After removing this file, we had no further issues.

Once we discovered the issue, it was straightforward to determine whether available memory had been exceeded by examining the binary image on the microcontroller. A better systematic approach would be to force the loader to abort if the static data size exceeded the available memory.

10.7.5 C Linker and Typing

One aspect that was difficult to debug involved an inconsistency between two autogenerated handlers that led to a function invocation with the wrong argument type. As previously mentioned, the Ivory code generates its own headers for communications primitives and user-level dispatch functions. Our AADL glue-code generator also generates headers for these functions. For periodic dispatch functions, there was an inconsistency in the type expected by the Ivory-generated code and the AADL prototype: the Ivory code expected the current time as an `uint32_t *` parameter, while the AADL code generated a `uint_32` parameter. Because each C file was using its own header, this was not detected at compile time, and, additionally, the C linker does not check argument types, so it was not detected at link time. The incorrect time values would cause the system to sporadically deadlock.

Another related issue involved time. We had three different programming systems: CAmkES, Trusted Build, and Ivory/Tower, that each have a built-in notion of time. In each case, times are ultimately compiled to C code, where time is represented as a 64-bit quantity. Unfortunately, the three environments did not initially agree on the granularity of time and its initial value, so calls between the different layers led to inconsistent and incorrect behaviors that were not compile-time checkable by the C language.

It is unfortunate and somewhat embarrassing that well-known problems in an antiquated linker technology can still cause substantial error. There are any number of simple fixes to these issues (e.g., embedding type information to allow discovery during link and adding unit types). However, such solutions do not have wide traction.

10.7.6 System Image Loading and Debugging

The PX4 board supports multiple methods to load and debug software onto it using the GNU debugger (gdb). Specifically, the Black Magic Probe can connect to the target microprocessor either via a JTAG interface or via a serial wire debugging port. The system software must be configured differently depending upon the debug interface. For example, the JTAG interface requires the initialization of a boot loader in memory. A mismatch between the configuration and the system load approach will cause segmentation faults. Unfortunately (and unexpectedly) different team members were using different mechanisms to connect to the PX4, so a binary built by one team would not work correctly for the other teams if they were using the other connection.

10.8 Modeling Concerns with AADL

When trying to generate and analyze system images, we identified several aspects of the system that were difficult to naturally model in AADL. These issues relate to the representation of interrupt requests, the representation of the target operating system, and the thread model in the context of a low-power processor with no protected memory. In addition, we present issues that we experienced in relation to the scheduling and communication mechanisms of the AADL. This section briefly examines each of the areas of difficulty.

10.8.1 IRQ/ISR Representation

Interrupt requests are asynchronous hardware-level signals that are raised when a device needs attention. They are serviced by interrupt service routines that act upon the processor interrupt. In most operating systems, ISRs are split into first-level interrupt handlers (FLIH) and second-level interrupt handlers (SLIH), where the job of the FLIH is to immediately dispatch on the interrupt and quickly record critical information that can be further processed by the SLIH, which is separately scheduled. In the eChronos RTOS, the FLIH is not a separate task, but executes in the context of the running thread. Further, the expectation is that the FLIH can perform only minimal processing and all aspects of the eChronos API are unsafe to use other than a special type of IRQ signal for the SLIH. In addition, some of the SMACCMcopter interrupts, such as the interrupt handler for the i^2c bus are time-sensitive and must run entirely in the FLIH, so we cannot easily bind the FLIH to a property within the SLIH thread.

Since FLIHs are not threads, it does not make sense to represent them as threads in AADL. On the other hand, they are not devices, which are defined as "dedicated hardware within the system" [27]; they are dispatched by the processor. Similarly, ISRs are not really features of the processor; they are software features. There is little guidance in the AADL documentation for representation of interrupt handlers.

For the time being, we are representing FLIHs as specialized devices that contain a custom dispatch function property. We add the FLIHs to the OS configuration file (as described in Section 7.2), and treat the SLIHs as "standard" tasks. This is, however, unsatisfying. Guidance on how to represent ISRs and IRQs (or a generalization of the device concept as it is currently presented in AADL) is necessary to correctly model this aspect of the system.

Recommendations Good support for IRQs and ISRs is necessary for correctly modeling low-level system behaviors as we find in the SMACCMpilot control software. There is very little guidance on this topic, however: the AADL standard [27] and textbook by Feiler et. al [5] do not describe how to correctly represent ISRs. Our use of AADL devices to represent ISRs allows us to generate code successfully, but seems inelegant and does not conform to the informal description of devices. An appropriate abstraction for ISRs (or, at least, an agreed upon convention for their representation) could ensure that our modeling style is compatible with the expectation of existing AADL tools.

10.8.2 Thread/Process Semantics

AADL supports a very rich specification of times in which to "freeze" inputs relative to thread dispatch. The default behavior is to freeze inputs at the time of dispatch, but this is 1.) expensive in terms of memory (since all data ports must be double buffered) and 2.) ambiguous as to the implementation with queued event-data ports. It is unclear as to whether this means that the entire queue is copied locally (to support "frozen" iteration over the queue), and how this relates to the designation of a full queue: is the queue 'full' for writers until the thread has completed its dispatch? Several additional supported input freeze behaviors are more exotic and seem quite difficult to implement in middleware. A clear accounting of the expectations on implementations for these different input freeze conditions is necessary to ensure they are implemented correctly. In addition, for resource-constrained processors, the user code can support an input-compute-output paradigm by coding style rather than having it enforced by the AADL-generated middleware, saving a significant amount of memory and time.

Additionally, for small embedded systems, a bare-bones RTOS such as eChronos and FreeRTOS do not have separate notions of thread and process; instead they just have *tasks*. There is no support for memory protection on these operating systems. In our system build tool, process boundaries are therefore meaningless; they only serve as a grouping mechanism for tasks.

Recommendations The first recommendation is a plea for a more rigorous description of input "freezing" in the AADL specification, and an affordance for it to be implemented by convention on the dispatchers rather than a requirement of AADL middleware; this would better support resource constrained hardware. Also, for hardware without memory protection, guidance for AADL modelers on appropriate notations for these kinds of systems (which are still architecturally interesting) is important.

10.8.3 Events and Schedulability

Another issue involves output event ports and schedulability. Currently, it is not possible to perform schedulability analysis in our flight model, in part because the properties governing the rate of dispatch for output event and event data ports are too coarse. AADL allows specification of an output rate property with the following type and defaults:

```
Rate_Spec : type record (
    Value_Range : range of aadlreal ;
    Rate_Unit : enumeration (PerSecond, PerDispatch);
    Rate_Distribution : Supported_Distributions;
);
```

This property allows some amount of analysis; unfortunately, for some threads in our models, there are several dispatchers for different event in-ports. Each of those dispatchers can trigger one from a subset of the output event ports; such relationships cannot be represented using the built-in property. This kind of relationship happens regularly in SMACCMpilot when performing input command processing. In this case, a decrypted ground station message may be one of several types; an input processing task determines the type and generates a single output event data message to the appropriate processing task.

One goal when choosing AADL as a system modeling language was to support analytic analysis of system schedulability; timing attacks are a known vector used by attackers to disrupt critical systems. However, the scheduling analysis tools we have found for AADL are either research prototypes or primarily support rate monotonic scheduling only. In a system involving a large number of sporadic events, such as the SMACCMpilot, we have been unable to find tools in which we feel we can trust the analysis.

Recommendations This problem is at the intersection of the structure of the architecture of SMACCMpilot and the capabilities of analysis tools. We have discussed this issue with the OSATE team at the Software Engineering Institute, where they are currently pursuing more robust and general schedulability analysis tools. We hope to collaborate with them on future analysis capabilities. We are also discussing ways to restructure the current SMACCMpilot architecture to make it more schedulability-analysis friendly.

10.9 Lessons Learned from ULB Application

Using research technologies such as those developed in the HACMS program is not without challenges, and this section addresses a number of lessons learned in applying the technologies to the ULB, and areas that if addressed could enhance the transitionability of the technologies to the defense industry. Note that we learned more lessons for some technologies than for others, but that should not be taken as any sort of indication of technology quality or value.

10.9.1 Ivory

From a CPS developer's perspective, the Ivory language presents as a subset of the C language. Some of the missing features, like malleable pointers, are removed for safety reasons. The rationale for other missing features, such as a complete for loop control (init, test and increment operations), is not as clear. Some of these missing features can make coding (especially porting of existing C code) more cumbersome and open to logic errors by the developer.

As with almost any language, interfacing with system libraries requires additional work. As Ivory generates C, the interfacing does not require any heavy lifting or interface complexities. However, it does require that you create Ivory definitions for anything that is actually used, so that type checking can complete and so appropriate code can be generated. In this case, the developer is responsible for correct translation and maintenance of the interface definitions. This could be less of a burden on the developer if Ivory could generate the required types from the existing C header files. One complication is that some concepts from C, like NULL or malleable pointers, are not valid in Ivory. Likewise, the creative use of the C preprocessor in some libraries makes determining the correct type for the final build difficult. A particular use case is in the definition of constants shared between Ivory and C. Ivory uses Haskell values which get turned into constant values in the generated C, as opposed to symbolic constants in header files, which limits their availability for use by the C code that uses the shared constants.

One way Ivory eliminates memory problems is to perform array indexing using modular arithmetic. Though this ensures that array bounds are never exceeded, it can lead to subtle and hard to find logic errors if this is not kept in mind. One way this is done is to define index types specific to each array size. This leads to a proliferation of types and can lead to copy/paste development with the potential for inconsistencies.

10.9.2 HCOL and Spiral

Since the component we were implementing was fairly straightforward and used only simple arithmetic and Boolean logic, we did not take advantage of the more advanced features of the technology. Likewise, by starting with HCOL, we did not take advantage of starting at a higher level of abstraction and reasoning about the algorithm at that level. Most of what was used on the ULB weight on skids component were the capabilities in Spiral for dealing with floating point instabilities and errors.

10.9.3 AADL and Trusted Build

One of the challenges with AADL is determining which parts of the specification to use. AADL is a fairly large and sophisticated specification, and for HACMS, we primarily used the software modeling elements at the level of process, ports, and threads, along with properties. HACMS, for example, did not use AADL flows. Some connections that were represented in AADL were not reflected in the glue code generated by AADL2RTOS. In particular, the network connection between the VSM and the FCC, and between the VSM and the ground station, were present in the AADL model. However on the VSM side, the network stack was implemented as an external component, while on the FCC side the VxWorks network stack was used, and it was invoked by "business logic" inside the component. This means that correctness of these connections is entirely up to the user, when the information to have these connections implemented in generated glue code is (or could be) present in the AADL model.

10.9.4 Tool Integration

There are a number of places where a lack of complete tool integration caused problems. The most significant of these is the lack of integration between Ivory and the AADL2RTOS translation in Trusted Build. To generate glue code, AADL2RTOS needs to know the entry points for the "business logic" of application components. Since the actual executed code for components implemented in Ivory is the compiled generated C code, the required entry points are the entry points of the generated C code. Unfortunately, these entry points are generated by the Ivory compiler. This results in an iterative process where the developer designs the system architecture, implements the components in Ivory, generates the C code from the Ivory implementations, extracts the entry points from the generated code, and then updates the model with the new entry points.

There is also the potential for problems when both the Ivory compiler and AADL2RTOS generate header files for the same entry points. In one case, this resulted in a buffer overflow. Ivory maintains array length values during compilation but these get erased when converting to a C char *. It is assumed that all of the code (headers and source) are generated by the Ivory compilation system which provides sanity checks. The Ivory generated function believed that a char* buffer in a struct was 300 bytes long and when clearing the array with a zeroing operation, writes 300 zeros to the buffer. In this case, AADL2RTOS also generated the header with the structure definition which believed that the buffer in the struct was 67 bytes long. This struct was allocated on the stack in one Ivory generated function and passed to the other Ivory generated function above. The allocated structure had a 67 byte array. The called function zeroed an extra 237 bytes, overwriting the stack including the return pointer value. In this case, the result was a segmentation fault. One potential solution would be for AADL2RTOS (or some other AADL-focused tool) to generate the Ivory definition files from the AADL model in a way that would ensure consistency and make sure that names and definitions matched.

10.9.5 Build Times

In some cases, build times for the current tools grew to the point where they had a significant impact. We found that with Ivory, a linear increase in code size turned into a seemingly exponential increase in compilation time. This was never noticed in Phase 2 as the VSM contained a smaller amount of code and was divided into small discrete components. During Phase 3, we converted the FCC which contained some large sections of code. Depending on what file was modified, a recompile could take upwards of 10 minutes.

In CAmkES, memory access is divided into 4k blocks. These blocks are declared separately in both the intermediate output and for the capdl tool. When a component has a large chuck of statically defined memory, such as the VSM component that stores all possible incoming STANAG waypoint data, this slows the CAmkES compilation process down.

Putting the two together, the recompilation time of the whole system became burdensome (upwards of 20 minutes in some cases), particularly on slower machines.

10.9.6 Debugging

Debugging using the HACMS technologies is challenging in several ways. Much of the executing code is autogenerated (AADL2RTOS glue code, CAmkES glue code, seL4 code, C generated from Ivory code), and this causes problems both in terms of understanding the generated code, and then moving from the generated code back to the original source, and then diagnosing the problem in the original source based on symptoms in the generated code. This is not a problem unique to HACMS, but it is exacerbated by the multiple scales and sources of code generation involved.

Another challenge is that some of the security features of HACMS technology can complicate debugging (for example memory separation can cause problems, and the use of authentication can make it difficult to restart part of the system).

10.9.7 Development Ecosystem

To move from a research environment to a production environment, there are a number of ecosystem considerations that would need to be addressed for some of these technologies to transition. This includes the usual concerns about documentation, training, process integration, and integration with other tools, such as quality management systems and life-cycle management systems. For cyber-physical/cyber military systems, there is usually a qualification/certification/accreditation requirement, and for DOD systems, there is the requirement for compliance with CNSSI-1253.

In a more particular sense, many of the HACMS technologies would benefit from enhancements to their own ecosystem. For example as a research language, Ivory has a limited set of libraries. With the inclusion array sizes in the type of an array, some sort of approach to standard libraries and/or polymorphism, would be beneficial. So for example, more extensive Ivory string libraries that doesn't require a new function for every string length. For seL4, it would be support for additional hardware, and hardware architectures. For example, the fact that the ULB flight control computer uses a PPC processor precludes the use of seL4.

11 Conclusion

Over the course of the HACMS program, a number of HACMS technologies were successfully applied, first to the SMACCMcopter research vehicle, and then to the Boeing Unmanned Little Bird. These technologies were successfully demonstrated on both aircraft in flight, including the successful defeat of attacks using a variety of common attack vectors. The SMACCMcopter was attacked via a remote data link, while the ULB was attack via a compromised USB maintenance device and a compromised supply chain. Additionally, a Red Team evaluation of the final HACMS aircraft, though still underway, has not found any significant exploitable vulnerability in the HACMS protected portions of the system.

The HACMS technologies were applied to the ULB by Boeing engineers (with the unstinting support of the technology researchers), including engineers from Boeing Defense Systems, not just those from Boeing Research and Technology. Together, this represents non-trivial evidence that these technologies are effective in improving system cybersecurity, can do so for cyber-physical systems without compromising the required real-time performance, and are usable by the developers of military systems.

12 References

[1] *20th USENIX Security Symposium, San Francisco, CA, USA, August 8-12, 2011, Proceedings.* USENIX Association, 2011.

[2] Andy Greenberg. Hackers remotely kill a jeep on the highway — with me in it, 2015. https://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/ (accessed 4-28-2017).

[3] Hugo Teso. Aircraft hacking: Practical aero series, 2013. https://conference.hitb.org/hitbsecconf2013ams/hugo-teso/ (accessed 4-28-2017).

[4] Kim Zetter. Feds say that banned researcher commandeered a plane, 2015. https://www.wired.com/2015/05/feds-say-banned-researcher-commandeered-plane/ (accessed 4-28-2017).

[5] Peter H. Feiler and David P. Gluch. *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language.* Addison-Wesley Professional, 1st edition, 2012.

[6] Jing Liu, John D. Backes, Darren D. Cofer, and Andrew Gacek. From design contracts to component requirements verification. In *NASA Formal Methods - 8th International Symposium, NFM 2016, Minneapolis, MN, USA, June 7-9, 2016, Proceedings*, pages 373–387, 2016.

[7] José Meseguer and Peter Csaba Ölveczky. Formalization and correctness of the pals architectural pattern for distributed real-time systems. *Theor. Comput. Sci.*, 451:1–37, September 2012.

[8] The Software Engineering Institute. OSATE: Plug-ins for front-end processing of AADL models, 2013.

[9] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM.

[10] NASA. Certware. `http://nasa.github.io/CertWare/`.

[11] Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, Stefan Savage, Karl Koscher, Alexei Czeskis, Franziska Roesner, and Tadayoshi Kohno. Comprehensive experimental analyses of automotive attack surfaces. In *USENIX Security*, 2011.

[12] Heartbleed. `http://heartbleed.com/`, February 2015.

[13] Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *USENIX Conference*, Berkeley, CA, USA, 2002. USENIX.

[14] Nicholas D. Matsakis and Felix S. Klock, II. The Rust language. *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, 34(3):103–104, October 2014.

[15] Tom Hawkins. Controlling hybrid vehicles with Haskell. Presentation. *Commercial Users of Functional Programming* (CUFP), 2008. Available at `http://cufp.galois.com/2008/schedule.html`.

[16] Lee Pike, Alwyn Goodloe, Robin Morisset, and Sebastian Niller. Copilot: A hard real-time runtime monitor. In *Runtime Verification (RV)*, volume 6418, pages 345–359. Springer, 2010.

[17] Emil Axelsson, Koen Claessen, Mary Sheeran, Josef Svenningsson, David Engdal, and Anders Persson. The design and implementation of Feldspar - an embedded language for digital signal processing. In *Implementation and Application of Functional Languages*, volume 6647 of *LNCS*, pages 121–136. Springer, 2011.

[18] Iavor S. Diatchki and Mark P. Jones. Strongly typed memory areas programming systems-level data structures in a functional language. In *Proceedings of the ACM SIGPLAN Workshop on Haskell*, pages 72–83. ACM, 2006.

[19] Iavor S. Diatchki, Mark P. Jones, and Rebekah Leslie. High-level views on low-level representations. In *Intl. Conference on Functional Programming*, pages 168–179. ACM, 2005.

[20] Sam Lindley and Conor McBride. Hasochism: The pleasure and pain of dependently typed haskell programming. In *Symposium on Haskell*, pages 81–92. ACM, 2013.

[21] Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. Giving haskell a promotion. In *Workshop on Types in Language Design and Implementation*, pages 53–66. ACM, 2012.

[22] Tom Schrijvers, Simon Peyton Jones, Manuel Chakravarty, and Martin Sulzmann. Type checking with open type functions. *Intl. Conference on Functional Programming*, pages 51–62, September 2008.

[23] John Launchbury and Simon L. Peyton Jones. Lazy functional state threads. pages 24–35, June 1994.

[24] Tim Sheard and Simon Peyton Jones. Template meta-programming for haskell. *SIGPLAN Notices*, 37(12):60–75, December 2002.

[25] FreeRTOS. Website `http://freertos.org/`. Retrieved Feb. 2014.

[26] eChronos. Website `http://ssrg.nicta.com.au/projects/TS/echronos`. Retrieved Feb. 2014.

[27] SAE-AS5506. *Architecture Analysis and Design Language*. SAE, Nov 2004.

[28] Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of LNCS. Springer, 2002.

[29] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *ACM Symposium on Operating Systems Principles*, pages 207–220, Big Sky, MT, USA, October 2009. ACM.

[30] Thomas Sewell, Simon Winwood, Peter Gammie, Toby Murray, June Andronick, and Gerwin Klein. seL4 enforces integrity. In Marko van Eekelen, Herman Geuvers, Julien Schmaltz, and Freek Wiedijk, editor, *International Conference on Interactive Theorem Proving*, pages 325–340, Nijmegen, The Netherlands, August 2011. Springer.

[31] Sidney Amani, June Andronick, Maksym Bortin, Corey Lewis, Rizkallah Christine, and Joseph Tuong. Complx: A verification framework for concurrent imperative programs. In *International Conference on Certified Programs and Proofs*, pages 138–150, Paris, France, December 2016.

[32] David Greenaway, Japheth Lim, June Andronick, and Gerwin Klein. Don't sweat the small stuff: Formal verification of C code without the pain. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 429–439, Edinburgh, UK, June 2014. ACM.

[33] Daniel Matichuk, Makarius Wenzel, and Toby Murray. An Isabelle proof method language. In Gerwin Klein and Ruben Gamboa, editor, *International Conference on Interactive Theorem Proving*, pages 390–405, Vienna, Austria, July 2014. Springer.

[34] Bernard Blackham, Yao Shi, Sudipta Chattopadhyay, Abhik Roychoudhury, and Gernot Heiser. Timing analysis of a protected operating system kernel. In *IEEE Real-Time Systems Symposium*, pages 339–348, Vienna, Austria, November 2011.

[35] Bernard Blackham, Yao Shi, and Gernot Heiser. Improving interrupt response time in a verifiable protected microkernel. In *EuroSys Conference*, pages 323–336, Bern, Switzerland, April 2012.

[36] Bernard Blackham and Gernot Heiser. Sequoll: a framework for model checking binaries. In Eduardo Tovar, editor, *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 97–106, Philadelphia, USA, April 2013.

[37] Bernard Blackham, Mark Liffiton, and Gernot Heiser. Trickle: automated infeasible path detection using all minimal unsatisfiable subsets. In Richard West, editor, *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 169–178, Berlin, Germany, April 2014.

[38] Thomas Sewell, Felix Kam, and Gernot Heiser. High-assurance timing analysis for a high-assurance real-time OS. *Real-Time Systems*, 2017. To appear.

[39] Thomas Sewell, Magnus Myreen, and Gerwin Klein. Translation validation for a verified OS kernel. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 471–481, Seattle, Washington, USA, June 2013. ACM.

[40] Ihor Kuz, Yan Liu, Ian Gorton, and Gernot Heiser. CAmkES: A component model for secure microkernel-based embedded systems. *Journal of Systems and Software Special Edition on Component-Based Software Engineering of Trustworthy Embedded Systems*, 80(5):687–699, May 2007.

[41] Matthew Fernandez, June Andronick, Gerwin Klein, and Ihor Kuz. Automated verification of RPC stub code. In *International Symposium on Formal Methods*, pages 273–290, Oslo, Norway, June 2015.

[42] June Andronick, Corey Lewis, and Carroll Morgan. Controlled owicki-gries concurrency: Reasoning about the preemptible eChronos embedded operating system. In *Workshop on Models for Formal Analysis of Real Systems (MARS 2015)*, pages 10–24, Suva, Fiji, November 2015.

[43] June Andronick, Corey Lewis, Daniel Matichuk, Carroll Morgan, and Christine Rizkallah. Proof of OS scheduling behavior in the presence of interrupt-induced concurrency. In *International Conference on Interactive Theorem Proving*, Nancy, France, August 2016.

[44] Leonid Ryzhyk, Peter Chubb, Ihor Kuz, Etienne Le Sueur, and Gernot Heiser. Automatic device driver synthesis with Termite. In *ACM Symposium on Operating Systems Principles*, pages 73–86, Big Sky, MT, US, October 2009.

[45] Sidney Amani, Alex Hixon, Zilin Chen, Christine Rizkallah, Peter Chubb, Liam O'Connor, Joel Beeren, Yutaka Nagashima, Japheth Lim, Thomas Sewell, Joseph Tuong, Gabriele Keller, Toby Murray, Gerwin Klein, and Gernot Heiser. Cogent: Verifying high-assurance file system implementations. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 175–188, Atlanta, GA, USA, April 2016.

[46] Christine Rizkallah, Japheth Lim, Yutaka Nagashima, Thomas Sewell, Zilin Chen, Liam O'Connor, Toby Murray, Gabriele Keller, and Gerwin Klein. A framework for the automatic formal verification of refinement from Cogent to C. In *International Conference on Interactive Theorem Proving*, Nancy, France, August 2016.

[47] Philip Wadler. Linear types can change the world! In *Programming Concepts and Methods*, 1990.

[48] Liam O'Connor, Zilin Chen, Christine Rizkallah, Sidney Amani, Japheth Lim, Toby Murray, Yutaka Nagashima, Thomas Sewell, and Gerwin Klein. Refinement through restraint: Bringing down the cost of verification. In *International Conference on Functional Programming*, Nara, Japan, September 2016.

[49] Sidney Amani and Toby Murray. Specifying a realistic file system. In *Workshop on Models for Formal Analysis of Real Systems*, pages 1–9, Suva, Fiji, November 2015.

[50] T. Bourke, R.J. van Glabbeek, and P. Höfner. Mechanizing a process algebra for network protocols. *Journal of Automated Reasoning*, 56(3):309–341, 2016.

[51] J. A. Bergstra and J. W. Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 37(1):77–121, 1985.

[52] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks*, 14:25–59, 1987.

[53] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, Englewood Cliffs, 1985.

[54] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[55] A. Fehnker, R. J. van Glabbeek, P. Höfner, A. McIver, M. Portmann, and W. L. Tan. A process algebra for wireless mesh networks used for modelling, verifying and analysing AODV. Technical Report 5513, NICTA, 2013.

[56] A. Fehnker, R. J. van Glabbeek, P. Höfner, A. K. McIver, M. Portmann, and W. L. Tan. A process algebra for wireless mesh networks. In H. Seidl, editor, *European Symposium on Programming* (ESOP '12), volume 7211 of LNCS, pages 295–315. Springer, 2012.

[57] David Hardin, Konrad Slind, Michael Whalen, and Tuan-Hung Pham. The Guardol language and verification system. In *Proceedings of the 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 7214 of *Lecture Notes in Computer Science*, pages 18–32, Tallinn, Estonia, March 2012.

[58] Standard interfaces of uav control system (ucs) for nato uav interoperablity. Technical Report STANAG 4586, NATO Standardization Agency, November 2012.

[59] Franz Franchetti, Aliaksei Sandryhaila, and Jeremy R Johnson. High assurance spiral. In *SPIE Defense+ Security*, pages 90911O–90911O. International Society for Optics and Photonics, June 2014.

[60] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[61] Lee Pike, Nis Wegmann, Sebastian Niller, and Alwyn Goodloe. Experience report: a do-it-yourself high-assurance compiler. In *Proceedings of the Intl. Conference on Functional Programming (ICFP)*. ACM, September 2012.

[62] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, LNCS, pages 168–176. Springer, 2004.

13 List of Acronyms

AADL	Architecture Analysis and Design Language
ADL	Architecture Description Language
ADT	Airborne Data Terminal
AES	Advanced Encryption Standard
AFRL	Air Force Research Laboratory
AGREE	Assume Guarantee Reasoning Environment
API	Application Program Interface
AST	Abstract Syntax Tree
AWN	Algebra for Wireless Networks
CAmkES	Component Architecture for Microkernel-based Embedded Systems
CAN	Controller Area Network
CBMC	C Bounded Model Checker
CFG	Control-Flow Graph
CORBA	Common Object Request Broker Architecture
COTS	Commercial of the Shelf
CPS	Cyber-Physical Systems
CPU	Central Processing Unit
CTL	Computation Tree Logic
CUCS	Common Unmanned Control System
DARPA	Defense Advanced Research Projects Agency
DFA	Deterministic Finite Automata
DMA	Direct Memory Access
DSL	Domain Specific Language
EDSL	Embedded Domain Specific Language
EGI	Enhanced GPS and Inertial
FAA	Federal Aviation Administration
FCC	Flight Control Computer
FLIH	First-Level Interrupt Handlers
GCM	Galois/Counter Mode
GCS	Ground Control Station
GDT	Ground Data Terminal
GHC	Glasgow Haskell Compiler
GPIO	General Purpose Input Output
HACMS	High-Assurance Cyber Military Systems
HET	Heavy Equipment Transporter
HCOL	Hybrid Control Operator Language
HOL	Higher-Order Logic
I2C	Inter-Integrated Circuit
IDE	Integrated Development Environment
IDL	Interface Definition Language
IOMMU	Input Output Memory Management Unit
ISO	International Organization for Standardization
ISR	Interrupt Service Routine

IRQ	Interrupt Request
ISR	Interrupt Service Routine
JTAG	Joint Test Action Group
LED	Light Emitting Diode
LOI	Level of Interoperability
MAC	Message Authentication Code
MBD	Model Based Development
MMU	Memory Management Unit
NVRAM	Non-Volatile Random Access Memory
OMG	Object Management Group
OS	Operating System
OSATE	Open Source AADL Tool Environment
OSI	Open Systems Interconnection
PALS	Physically Asynchronous Logically Synchronous
PWM	Pulse Width Modulation
RFE	Radio Frequency Equipment
RPC	Remote Procedure Call
RT	Real Time
RTOS	Real Time Operating System
SMACCM	Secure Mathematically-Assured Composition of Control Models
SMT	Satisfiability Modulo Theories
SLIH	Second-Level Interrupt Handlers
SPI	Serial Peripheral Interface bus
SRB	Safety Review Board
STANAG	Standard Agreement
TCB	Thread Control Block
TLB	Translation Lookaside Buffer
TRB	Technical Review Board
UART	Universal Asynchronous Receiver/Transmitter
UAV	Unmanned Air Vehicle
UDP	User Datagram Protocol
USAF	United States Air Force
ULB	Unmanned Little Bird
WCET	Worst Case Execution Time
VCPU	Virtual Central Processing Unit
VM	Virtual Machine
VMM	Virtual Machine Monitor
VMS	Vehicle Management System
VSM	Vehicle Specific Module